Meaningful Method Names

Doctoral dissertation by

Einar W. Høst

Submitted to the Faculty of Mathematics and
Natural Sciences at the University of Oslo
in partial fulfillment of the requirements for
the degree Philosophiae Doctor in Computer Science

November 2010

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1044

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Unipub.
The thesis is produced by Unipub merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Abstract

We build computer programs by creating named abstractions, aggregations of behaviour that can be invoked by referring to the name alone. Abstractions can be nested, meaning we can construct new, more powerful abstractions that use more primitive abstractions. Thus we can start from tiny blocks of behaviour and build arbitrarily complex systems. For this to work, however, the abstractions must be sound — in other words, the names must suit the behaviour they represent. Otherwise our tower of abstractions will collapse. Hence we see the crucial importance of naming in programming.

Despite this importance, programmers almost completely lack tools to assist them. The computer treats names as arbitrary, allowing for sloppy and inconsistent naming. Tool support for good naming would be beneficial for many reasons. Most obviously, it would help create programs that are easier to understand, and hence easier to maintain. A secondary, but equally important, effect is that good naming and good design go together. In other words, good naming strengthens the tower of abstractions.

In this thesis, we show that the method names used in Java programs are far from arbitrary. They are *meaningful* in a sense that relates to the behaviour they represent. By analysing the implementation of methods in real-world Java programs, we can approximate the meaning of names and gain a deeper understanding of key aspects of naming in Java. For instance, we show that it is feasible to create a tool to discover *naming bugs* in Java programs — methods that have been improperly named. Our analyses are completely mechanical, meaning that they require no human supervision.

Acknowledgements

First of all, I would like to thank my main supervisor, Bjarte M. Østvold, for providing motivation, support, inspiring discussions and never-faltering faith in the research. You are an excellent supervisor — it has been invaluable to me that you always kept your door open, always found time and energy to listen or contribute ideas. Working with you has been both educational and great fun. I would also like to thank my co-supervisor Gerardo Schneider for kind assistance and cooperation in all practical matters, as well as valuable proofreading and comments.

The main part of the work presented in this thesis was done while I was employed as a PhD fellow at Norsk Regnesentral. I would like to thank the head of the DART department, Åsmund Skomedal, for having enough faith in me to hire me. I also appreciate the kind faces of the rest of the DART employees. Thank you to professor Barbara G. Ryder for inviting me to Rutgers during my PhD fellowship, a trip that greatly expanded my horizon and taught me some valuable lessons. I would also like to thank Jan Wloka for many interesting discussions, both professional and personal, over coffee ranging from the excellent to the abysmal. I learned much from you.

My work at Norsk Regnesentral was supported by a grant from the Research Council of Norway through the RSE-SIP project. I am grateful to the staff at the Department of Informatics at the University of Oslo for extending my PhD contract so that I have been able to complete my work. I would also like to thank my current employer, Computas, for flexibility and support during the final phase of my work.

Thank you mum and dad for your endless support and understanding. You have taught me the value of knowledge and learning, as well as the joy in working to accomplish something. I am proud and grateful to have been raised in that tradition. Finally, my deepest thanks to my wonderful family — my ever-optimistic and positive wife Line and my two amazing children Astrid and Sigurd — for filling my life with light, laughter and love. You make every day meaningful and valuable. Thank you.

Contents

Part I

Overview

Chapter 1

Introduction

> The limits of my language mean the limits of my world.
> - Ludwig Wittgenstein.

In computer science, the world is the artificial world of the computer. This world is shaped by humans, who use artificial languages to write programs to make it ever more sophisticated. In this respect, Wittgenstein's quote is especially appropriate: the richness of the artificial world of the computer is determined by what we can express using programming languages.

Programming languages lend much of their power from the ability to create meaningful abstractions. Abstractions are useful because they allow programmers to create a more powerful language in which they can express the solution to a problem. In his 1998 OOPSLA keynote, Steele referred to this as *growing a language* [38]. In a sense, creating meaningful abstractions is *the* core programmer activity. At the same time, we don't seem to understand it very well [41]. For instance, we lack the ability to check whether or not an abstraction is meaningful. Indeed, we don't even have suitable criteria for making such claims.

Typically, abstractions are built by grouping together a sequence of instructions for the computer (with a formal semantics in terms of the low-level operations the computer should perform), and providing a label, a name, for that sequence of instructions. In most conventional programming languages, the basic unit of abstraction is the *method*[1]. The name of a method acts as an informal semantic annotation for the implementation. This annotation has no formal function besides acting as a lookup-mechanism: in text-based programming languages, the method name is used to find the correct sequence of instructions to execute. In principle, however, the annotation could have a formal meaning enforced by the computer.

While the computer treats the method name as arbitrary in all programming languages we are aware of, we also know that it rarely is arbitrary in practice. Rather, programmers choose as meaningful names as they can for their methods, since this is needed for the abstractions to be sound and solid. It is very difficult indeed to read a program without meaningful names — which is why every program obfuscator will make sure to scramble names.

[1] Also: function, procedure.

1.1 Research Goals

The overall goal of this thesis is to show that:

> *The meaning of method names can be derived from programming practice.*

While this is a straightforward statement of ambition, we must explore it further for it to become truly meaningful. Indeed, all the salient terms in the statement warrant investigation and interpretation. First, we must investigate what *method names* are. Second, we must establish a theory of *meaning* that applies to the method names. Third, we must provide some interpretation of what *programming practice* is. Finally, we must specify what it means to *derive* the meaning from practice.

These considerations lead us to formulate the following research goals:

- G1 Show that a significant part of method names in the real world follows patterns that can be identified and described.
- G2 Forge a link between the informal semantics indicated by method names and the formal semantics of method implementations.
- G3 Investigate key aspects of naming by inspecting how method names are applied to implementations in practice.

These research goals presuppose that we have some way of tapping into what is representative of programming practice in the real world. A suitable corpus of software applications must therefore be considered a prerequisite for all the goals.

1.2 Summary of contributions

Here we summarise our contributions towards the research goals and the thesis statement. We believe that the informal meaning of method names relates to the formal semantics of implementations in such a way that we can usefully approximate the informal meaning of names by analysing the implementations they represent. In order to do so, we must build a suitable framework for analysis, gather a representative corpus of programs to analyse, and show that by applying the analysis, we can indeed answer interesting questions about what method names *mean* in terms of implementations, and how they relate to one another.

Name patterns. We show that method names are phrases that exhibit simple grammatical structure — typically, a method name is a command consisting of a verb, often followed by a noun. Furthermore, we develop a simple notation that allows us to abstract over concrete method names to form name patterns. We also measure the prevalence of the most common patterns. We find that name patterns are useful because they allow us to focus on the generic, domain-independent vocabulary of programmers.

Usage semantics. We supply a formal definition of a *usage semantics* for method names. The usage semantics reflects the formal semantics of the methods implementations themselves. In other words, we root the meaning of method names in how the methods are typically implemented. To abstract over the concrete implementations, we use a notion of semantic profiles. This notion greatly simplifies comparison of implementations and recognition of essential similarity in the face of many superficial differences.

Understanding naming. We demonstrate the usefulness of our approach by illuminating key aspects of naming. We characterise each commonly used method name by providing an automatically generated textual description based on typical implementation features. We also measure precision and consistency in naming. Furthermore, we identify names that are similar to each other, and mechanically detect "naming bugs" in many Java programs.

Chapter 2

Research method

We outlined in Chapter 1 the problem domain we want to investigate; here we consider how the investigation should be conducted. Clearly, we would like the investigation to qualify as *scientific*. However, this is non-trivial since we share Chalmers' belief that no *universal* account for science or scientific method as such can be given [7]. We nevertheless assume that some sort of *scientific research method* is both attainable and desirable for the investigation. We take here the position of what Worrall calls structural realism [43], meaning that we believe that scientific theories can capture some essence about reality, without necessarily being "true" in the unattainable objective sense. In particular, scientific theories are suitable to describe the structural relationships between entities. Worrall's account of structural realism is ambiguous with respect to the entities themselves, which could be considered beyond the grasp of scientific inquiry, or even illusory.

Philosophically, computer science is interesting because man-made artefacts constitute a major part of the field of inquiry – with man constituting the other major part. We note three dimensions to consider when discussing computer science research: *method*, *narrative* and *relevance*.

By method, we mean the way in which the researcher conducts research. In lieu of a universally applicable approach to "scientific method", generic frameworks or models describing a work process for scientific conduct have been suggested [15, 37]. These frameworks cannot by themselves guarantee proper scientific method. Rather, they can be useful in planning research work, or to enable accounting for the "scientificity" of the work leading to the research narrative.

By narrative, we mean the way in which the researcher embodies the research, and conveys it to the rest of the scientific community. This is as important as the research method itself. Russell reminds us that science is about description, not logic or causation or any other naive notion we have about necessity [33]. We must therefore consider what *scientific narrative* is. Clearly, we have expectations regarding honesty, accountability, transparency, completeness and so forth. We also expect a certain style of presentation, typically "fact-oriented" and without rhetorical tricks such as suggestive language. Arguably, however, these demands can potentially act as hindrance to radically new approaches or ideas. Whereas in-paradigm research can rely on conventional opinions and be modelled after existing papers, paradigm-breaking research may have to convince its readers about new kinds of facts or justifications. To compensate for this, some computer science conferences have started inviting "big idea" papers, essays (a loosening of the requirements for scientific narrative) and so forth. The Onward!

conference series, for instance, invites papers on "not so well-proven but well-argued ideas". This is a direct tipping of the equilibrium away from the strictly scientific.

By relevance, we mean the degree to which the narrative is of interest to anyone outside the scientific community. Of course, scientific work need not be relevant (useful) in order to be valuable: this is the distinction between basic and applied research. On the other hand, many researchers would *like* for their narrative to be influential in "the real world" — whether for altruistic or egotistical motives. Hence, there is an incentive for the researcher to make the narrative as appealing as possible to outsiders, so as to appear relevant. The desire for perceived relevance could be in conflict with the goal of a sober, scientific narrative, in that the researcher may be tempted to employ rhetorical tricks to compensate for lackluster results.

2.1 Research on programming

The research topic of this thesis is *programming*, the activity by which we are able to create and expand the artificial world that is the modern computer in operation. The study of programming is often limited to research on *programming languages*. Programming languages are peculiar languages, since they are devised to directly bridge the gap between man and man-made artefact. As such, research on programming languages is a particularly poignant example of the dual nature of computer science research. Yet we are primarily interested in the languages as a *means of programming*: we would like research on programming languages to improve the way in which we program!

As objects for scientific inquiry, man and artefact (computer) are radically different. Man is notoriously unpredictable, whereas the computer is designed to be deterministic. In a sense, it is easier to do "obviously scientific work" on the computer side of the equation. It is clearly desirable that features of a programming language are rooted in a theoretical framework that allows for the verification of soundness and consistency. Indeed, computer scientists have enjoyed great success by thorough investigation of the mathematical, logical and semantic properties of language constructs, yielding great benefits for instance in the form of unambiguous, consistent programming languages, ever better compilers, methods for formal verification of correctness and so forth.

The human side of the equation is more problematic, since research involving humans is both difficult and expensive. But we need this research to verify — or at least make probable — that our programming language research is indeed improving the way we program. Otherwise, we run the risk that our innovations are intellectual exercises with little practical value.

2.2 Narrative and relevance: Influencing programmers

It is common among industry practitioners to complain that computer science researchers pay too little attention to "the real world". Scientists are accused of locking themselves up in the "ivory tower", a somewhat derogatory term that designates a distant place where intellectuals engage in pursuits that are disconnected from the practical concerns of everyday life. By contrast, the same practitioners use a metaphor

as severe as *war* to illustrate the harshness of the reality in which they find themselves, speaking of "life in the trenches" and "veterans" recounting "war stories" from both failed and successful projects.

When aiming for *relevance* of research, researchers must create a narrative that is compelling enough to compete with such war stories. The primary selling point of war stories is the credibility that stems from the real world. At the same time, war stories are unscientific, anecdotal and suggestive by nature. They have serious shortcomings when held against scientific standards. This indicates that learning from war stories is problematic, that generalisation from story to principle may be unsound. Scientific narratives in the form of research papers can compete, if they can overcome the problem of being removed from the real world. Ironically, a research paper on programming that is limited to toy examples suffers from similar problems as war stories: it is hard to generalise from the results.

2.3 Method: Empirical studies

The preceding discussion indicates that *empirical studies* are suitable for research on programming, both as a means to study programming as a human activity, and to recount a scientific narrative that is relevant to practitioners. Arguably, innovation in programming languages is arbitrary and essentially un-scientific unless it is guided by an understanding of how programming with current languages is conducted. This understanding can only be gained from empirical studies.

Of course, this idea is not new: in their 1975 ACM Turing Award lecture, Newell and Simon note that computer science is an empirical discipline [29]. However, they also note that "some of its unique forms of observation and experience do not fit a narrow stereotype of the experimental method". Indeed, it is both costly and difficult to set up controlled and reproducable experiments in programming, since there are so many contributing factors and sources of uncertainty — in particular, the human programmers themselves. Presumably this is why, twenty years later, Tichy et al. [39] find that computer science is still sorely lacking in the use of experiments.

However, there may be questions about programming that we seek answers to empirically, without conducting experiments. Rather than studying the programmer at work, we can study the artifacts he has created. This study is facilitated by the rise of the open-source software movement, which has made available a rich and varied "body of literature" written in programming languages. Unsurprisingly, we observe a corresponding increased interest in research on software artifacts, in particular in the software repository mining community. This community holds promise to fill an arguable gap in computer science research, by enabling research on programming that takes into account the programmer as well as the machine. In his keynote at the Mining Software Repositories 2010 conference, James Herbsleb argued that the conference should be ambitious enough to aim at forging a science of socio-technical behaviour.

2.4 The research method of this thesis

We discuss the research method employed in this thesis in terms of the four phases described by Glass [15]. The phases are *the informational phase* (gathering of information through reflection and/or surveys of literature), *the propositional phase* (proposition

of a model or approach), *the analytical phase* (exploration of the proposition) and *the evaluative phase* (evaluation of analytic findings). Note that we use these phases for a structured discussion of the research work, rather than as an absolute presentation of the chronology of the work. In practice, the actual work was not as neatly structured into phrases as the presentation might suggest; in particular, analytical findings would often cause us to revise details of our proposed model.

2.4.1 Informational phase: Informal meaning

The informational phase coincided with a hermeneutical process of identifying and formulating research goals. As a backdrop to our reflections, we conducted a relatively broad survey of research papers on patterns, tools for programmer assistance, code analysis, software artifact search, natural language programming and so forth. During this process, the idea crystallised of investigating the relationship between the informal meaning of natural language names and the formal meaning of the programming language constructs. Following this idea further lead to identifying a philosophy of how meaning arises in traditional use of natural languages, and transferring this to the realm of programming, where natural language expressions are mapped onto sequences of programming language instructions.

2.4.2 Propositional phase: Abstract semantics

Once we knew the overall goal for our research — that is, to investigate the meaning of method names — we ventured to create a *model* of programs that would facilitate this investigation. Any model is a description of reality that highlights some aspects while ignoring others. The essential problem that we needed to overcome was that it is generally *hard* to compare programs, *hard* even to compare individual methods. Furthermore, we identified the need to forge a link between the informal semantics of method names and the formal semantics of the programming language.

In this thesis, we propose to use abstract semantics and statistical considerations to accomplish these goals. The abstract semantics is used to capture "the essence" of a method implementation, while filtering out "the accidental". We use statistics to correlate names and implementation characteristics. This is rooted in our philosophy of how the meaning of a word arises in natural language: it simply stems from how the word is used in practice. Hence the proposition indicates the need for a *software corpus* representative of programming practice in the real world.

We therefore created a theoretical framework that ties the informal method name semantics to the formal semantics of the method implementations themselves. The central notion in this framework is a coarse-grained model of implementation semantics that abstracts over the formal semantics. The salient feature of this model is that it enables *comparison* between method bodies as semantic objects.

2.4.3 Analytical phase: Answering questions

The analytical phase consisted of three major parts: 1) coming up with interesting and suitable research questions to investigate using our proposed approach, 2) creating the software corpus to be used as data for our analyses when seeking answers to these

questions, and 3) defining and performing the actual analyses. To come up with interesting research questions is easy; however, they must also be suitable for investigation using our approach.

The availability of open-source software makes creating a software corpus easier. However, there are still many issues to consider when creating a corpus. First, the corpus needs to be large and diverse enough, so as to represent a reasonable cross-section of programming practice. Second, boundaries between programs can be weak, due to heavy use of libraries and frameworks. Care must be taken to avoid analysing the same software artifact more than once. Third, code generation can lead to software artifacts that are "unnatural" (that is, not representative of human programming), and that can skew deductions about programming practice due to duplication.

Our basic approach of using abstract semantics and corpus analysis to seek answers to research questions is common for all our research. However, what is "essential" for a method implementation is somewhat dependent upon the actual question being answered. For some questions, such as how to best describe a given verb in a method name (see Chapter 7), relatively broad characterisations of implementations may be useful — such as whether or not it is common for the implementation to contain a loop. For other questions, such as which implementation features are improper for methods of a given name (see Chapter 9), narrow characterisations are more appropriate — such as whether or not the method returns an object it has created. In each paper, we therefore tailor the abstract semantics to the questions we seek to answer.

2.4.4 Evaluational phase: Hypothesis testing

We identified the need for two levels of evaluation in our work. At the overall thesis level, we need to evaluate the adequacy of our propositions. A model is useful to the extent that it gives useful answers to the questions we pose. Also, we want the answers not to be misleading. While we believe that the model has proven its capabilities through the results presented in the research papers, we also acknowledge that it has some limitations. In Section 5.2, we summarise our critique of the model.

At a lower level, we need to evaluate what happens when we answer the individual questions in the various papers. In our case, this is particularly important, since we do unsound analysis — that is, analysis that can yield false positives. Since we deal with informal semantics, modelling something that is not formalised, implicitly understood by programmers, we need to rely on subjective judgement when performing this evaluation. In other words, we need a human oracle to measure the performance of our approach.

In the individual research papers, we present our own subjective judgement of the accuracy of our results. Where possible, we quantify the number of false positives — however, this number is still subject to our own judgement. Unfortunately, given the nature of the problem, we lack any objective measure with which to compare our work.

Chapter 3

Problem analysis

We noted in Chapter 1 that *named abstractions* are the building blocks used by programmers when constructing programs. Modern programming languages are typically accompanied by a rich set of existing building blocks, in the form of application programming interfaces (APIs) and frameworks. Programmers build their own named abstractions on top of the pre-existing ones. In an object-oriented language like Java, the named abstractions are methods and objects. The names act as informal annotations of semantics; unimportant to the computer, but crucial to the programmer. It is the programmer's main defence against the overwhelming number of semantic levels involved in programming [10]. The resulting program is a complex structure, consisting of interrelated abstractions that have both formal semantics (as specified by the programming language) and informal semantics (as indicated by the annotations or names). The formal and informal semantics of abstractions are not independent; they mirror each other. As programmers, we share the experience of changing a method implementation to better suit a method name, and of updating a method name to reflect a change in the implementation. However, the relationship between formal and informal semantics is not well understood beyond the simple intuition that they should somehow "harmonise". We aim at understanding how these two layers of meaning relate to one another.

We restrict our discussion of named abstractions to *methods* in the Java programming language. In our perspective, the method form the cornerstone of abstraction in programming, since it represents the smallest unit of named, aggregated behaviour. We find it likely that the analysis that follows holds for other object-oriented languages as well, since naming patterns and semantics are similar. For languages belonging to different paradigms, it may have to be adjusted. However, there is a similar relationship between names and implementations in functional languages as well, although the naming patterns and semantics may be different.

3.1 A pragmatic theory of meaning

To understand the relationship between names and implementation semantics, we must first establish a theory of how natural language expressions become meaningful in the first place. Inspired by Wittgenstein [42], we believe that meaning derives from practice: the meaning of an expression is merely a summary of previous uses of that expression. It follows that expressions are only meaningful if they are used consistently.

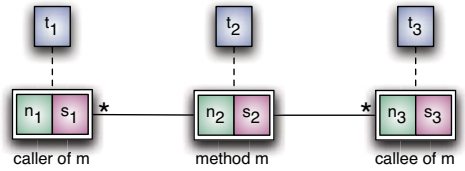


Figure 3.1: A method acts as a focal point of meaning.

Furthermore, it is clear that we need experience with how an expression is conventionally used in order to meaningfully use the expression ourselves.

When we use an expression in natural language, many dimensions are at play: the expression itself, the context in which it is used (which often includes other expressions), what the user meant by the expression, and the history of previous uses of the expression. According to Wittgenstein, it is this history that is the meaning of the expression. This may or may not coincide with what the user really meant. It is worth noting that in the context of programming languages, there are two ways in which we can "use" an expression involving a method: as implementer or caller. Use-as-caller is very similar to conventional use of expressions in natural languages. However, use-as-implementer is interesting, because it provides us with an explicit specification of what the user meant by the expression: the actual implementation, which has a formal semantics. This is an alternative source of meaning for the expression, and one that can be in harmony or conflict with the history of previous uses. (When a programmer names an abstraction, we have an instance of use-as-implementer.)

The naming of an abstraction is really just an instance of juxtaposition of label (the name) and semantic object (the implementation). As a consequence of the juxtaposition, the semantics of the object "rubs off" on the label that is used. Of course, a single juxtaposition of label and object is insufficient to establish a stable link between the two. Rather, the meaning of a label stems from consistency in juxtaposition: we repeatedly use the same label for the same object.

3.2 Informal meaning in programs

We can use a *call graph* [16] as a starting point for our investigation. A call graph is a common representation of a program, where the nodes represent methods, and the edges represent calling relationships between methods. The method names act as informal annotations of the semantics of a method. The call graph is an interesting object of study for our purposes, since it exposes formal relationships between named entities, and relates names to entities with formal semantics.

Fig. 3.1 shows how a method may act as a focal point of meaning in a call graph. All methods m have a name n, a formal semantics s and an associated type t. In addition, the call graph will reveal that m has a set C_r of zero-to-many *callers* (other methods that call m), as well as a set C_e of zero-to-many *callees* (other methods called by m). From this, we can identify all kinds of interesting relationships to investigate, such as between n and s, between n and the names of the callers, between n and the names of the callees, between s and the semantics of the callers, between s and the

semantics of the callees, and so forth. These relationships are all established by method calls present in the method implementations. Method calls are particularly interesting for our purposes, since they represent *uses* of names. In addition, we note that types serve to create another set of meaningful relationships between methods.

What is the method name? Ironically, the term *method name* is something of a misnomer. In *The Java Programmer's Phrase Book* (Chapter 8), we note that method names play three quite different roles in programs: a technical role (to allow lookup of methods), a mnemonic role (to aid human memory) and a semantic role (to reflect implementation). The terms we use, *method name* and *method identifier*, both fail to capture these three roles adequately. The term *identifier* seems to correspond to the technical role, whereas the term *name* corresponds to the mnemonic role. However, there is no term corresponding to the third role, which is the one we are most interested in here.

We can plainly see that a method name is not simply a name, since a typical method name has grammatical structure. Rather, a method name is a natural language phrase that act as a *description* of what a method does. This description simultaneously acts as a *promise* made to callers of the method. Hence there needs to be an implicit contract of *accountability* in naming between programmers. In other words, the method must hold true to its promise: do as the name indicates, and nothing else.

What is the method semantics? We have stressed the dual nature of the method as a named abstraction, where the name acts as a promise to what the implementation does. The implementation is quite simply the sequence of bytecode instructions found in the method body. However, we know from the very existence of a call graph that methods often call other methods. In an interprocedural perspective, therefore, the full semantics of a method must recursively subsume the semantics of any callee. Hence the promise made in the name of a method really encompasses a sub call graph with the original method as the root.

3.3 Interpretation of meaning

Any method m in a program lives in the intersection of force fields of meaning stemming from the many relationships implied by Fig. 3.1: between expectations from the callers, promises from the callees, and its own implementation. Taken together, we see a web of meaningful relationships stemming from the call graph. The problem, then, becomes how to interpret or make sense of this web.

Our pragmatic theory of meaning indicates some need for *aggregation* in order to understand how names relate to implementations. If meaning emerges from a history of consistently applying the same label (method name) to the same semantic object (method semantics), then we need a way to identify instances where the same label has been applied, as well as instances where the same semantic object has been labelled. Furthermore, we need to be able to aggregate across programs. A single text in natural language is insufficient to establish the conventional meaning of words in that text; similarly, we must investigate naming across many programs to understand what the names mean.

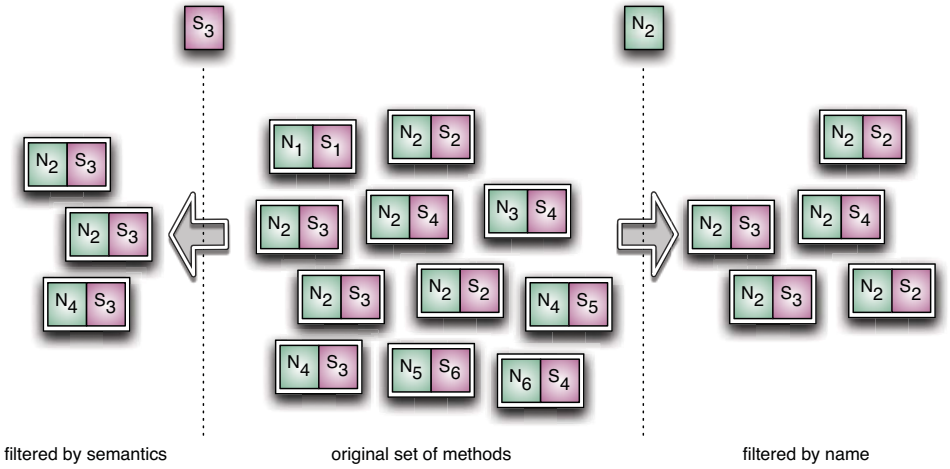


Figure 3.2: Selecting nodes.

We can define *selectors* to identify substructures in a call graph that are equal in some respect. These selectors can be very simple or arbitrarily complex. An example of a simple selector is one that selects methods based on a naming pattern. A more sophisticated selector may attempt to identify design patterns in the call graph. Fig. 3.2 shows the effect of two ways of filtering or selecting groups of nodes from a graph, based on semantics and name, respectively.

3.4 Ambitions

In Section 1.1 we presented three research goals to support our thesis statement. We now elaborate on each research goal somewhat, and present our ambitions with respect to each goal.

3.4.1 Goal G1: Name patterns

Underlying this thesis is the claim that method names are not arbitrary labels; they are meaningful natural language expressions, composed of one or more words. If this is the case, we must be able to find traces of grammatical structure in method names, just as we do in expressions in conventional natural languages like English. Indeed, implicitly in goal G1 is a notion that there exists a "generic" natural language used by Java programmers when writing programs. We seek to uncover and understand this language. This requires us to:

- **Decompose the names.** If method names are multiword natural language expressions encoded in a single string of characters, we must parse and decompose the names to recover the individual words in the expressions.
- **Analyse the names.** When we have recovered the individual words we must analyse the name as a natural language expression. We should then look for

patterns by finding what varies and what remains stable between names.

- **Abstract over names.** We must provide an abstraction over method names, so as to highlight the stable parts of method names while allowing for variation. This will in turn enable us to express naming patterns.

- **Identify name patterns.** We should verify our claims by identifying, describing and counting occurrences of naming patterns in real-world Java applications.

3.4.2 Goal G2: Usage semantics

The purpose of this goal is to enable us to capture the meaning embedded in method names. Arguably, this is the pivotal requirement for our thesis. To forge a link between method names and implementations, we must define a *formal semantics for method names* to approximate the *informal* meaning of method names as understood by programmers. The definition must reflect our pragmatic view of how meaning arises — or rather, how names accumulate meaning through use. Hence the semantics must be a *usage semantics*, one that reflects how names and implementations are paired in the real world of Java programming. In other words, we are exploiting the use-as-implementer view as discussed in Section 3.1.

In order to be useful, the semantics we define for method names must overcome the difficulties in comparing method implementations that arise due to accidental or non-essential differences. We must therefore provide a suitable and flexible abstraction over method implementations that enables us to focus on the essential attributes of the implementations and disregard the rest.

3.4.3 Goal G3: Understanding naming

Ultimately, the usage semantics of research goal G2 is only useful if it allows us to illuminate key aspects of the meaning of names. In particular, the approach should prove its worth by allowing us to *characterise* methods with a given name, deem whether or not an implementation is *suitable* for the name, identify other names that are *similar*, as well as measure the *precision* and *consistency* of the name. Below we provide rationale for each of these aspects.

Characterisation. To know what a name means, we must be able to articulate its meaning. We note that to characterise a name implies distinguishing it from other names. To say what something is, is also to say what it is not. Of course, we find the same phenomenon in natural languages: when we seek to describe the meaning of words, we must ultimately do so circularly, in terms of other words that may be synonyms or antonyms. We aim to provide meaningful textual descriptions for all the commonly used method names. The descriptions should be non-trivial, ring true with Java programmers, and feel like a reasonable summary of the salient features of typical method implementations.

Suitability. To know what a name means, we must be able to say when something is in violation of that meaning. A violation of meaning occurs when there is conflict between the name and implementation of a method. In other words, the name

and implementation should *suit* each other — otherwise one or the other should be corrected. In a sense, the notion of suitability is just a different perspective on the characterisation of a name. For each name, we can imagine a boundary between meaning and not-meaning. What we think of as the *meaning* of a name is really just one side of the boundary: we cannot say what something means without the backdrop of an implicit not-meaning. We aim to associate rules for implementation with method names, providing a clear boundary that lets us identify violations of meaning. We should demonstrate this by finding examples where implementation rules are violated in real-world Java applications.

Similarity. In general, an important aspect of a name's meaning is how it relates to the meaning of other names. Similarity is particularly interesting relationship since it involves the degree to which the meaning of two names *overlap*. We should be able to judge two method names as being similar or dissimilar, and to order names according to similarity. For names that are very similar, another question arises: whether or not the names can be considered to be *synonymous*.

Precision. Precision can be interpreted as *how much* meaning a name has. A precise name tends to be implemented consistently in the same way. As such, precision is a partial measure of the quality of a name. In general, we prefer names that are clear, well-understood and useful. As noted by Blackwell [5], a name such as **processData** is practically useless since it conveys no meaning[1] — reading the name gives us no intuition about what the method does. We aim to quantify the precision of names, thus being able to judge whether or not a method name has a clear convention with regards to implementation.

Consistency. Consistency of naming means applying the same label to the same object; in other words, naming our implementations consistently. Interestingly, we can only do so if the implementation has a precise meaning. Hence there is a duality between precision and consistency: precision in naming requires consistency in implementation, consistency in naming requires precision in implementation. We aim to quantify how consistently an implementation is named, and to identify implementations that are problematic in the sense that they are practically "unnameable".

3.4.4 Prerequisite: Representative corpus

To meet our ambitions with respect to the research goals, we need a corpus of real-world Java applications that is *representative* of Java programming. To be representative, the corpus must meet requirements with respect to:

- **Size.** The corpus must be large enough. This is to ensure that idiosyncrasies of individual applications are levelled out.
- **Variety.** The corpus must consist of applications from a wide range of domains. This is to ensure that we cover all kinds of Java programming; there might be stylistic or conventional differences between different types of applications.

[1] So-called *functors* or function objects are a notable exception; the convention is to give the functor a single public method with a generic name like **execute**. The meaning missing from the method is often found in the type name, however.

- **Recognition.** The corpus should consist of well-known Java applications. We should strive to avoid obvious omissions. Well-known applications are interesting because they are examples of "successful" Java programming and might be influential with respect to naming.

We also need to address potential problems with the data in the corpus. In particular, we are sensitive to the fact that the corpus is likely to include some very commonly used libraries. Unfortunately, there is no simple way of providing a clear boundary around a single Java application. Measures must be taken to ensure that each library is included only once. Code generation is another potential problem; we should take precautions so as to diminish any negative impact of this.

Chapter 4

State of the art

In Chapter 3, we presented the problem of how identifiers become meaningful. We claimed that the program graph, annotated with names, is the principal object to examine in order to understand the problem better. Using the name-annotated graph, we will seek to bridge the gap between the *programmer language* we find in the natural-language expressions encoded in the names, and the formal semantics of the *program structure*. In this chapter, therefore, we survey the state of the art with respect to *exploring programmer language* and *finding meaningful artefacts in program structure*, as well as attempts to combine the two.

We narrow our discussion of programmer language down to the language encoded in identifiers. Encoded in identifiers such as method names, we find natural language expressions. In Java, the expressions are subject to some syntactic limitations; a method name may not contain white space or punctuation, for instance. This can sometimes cause problems, since we typically rely on punctuation (in particular quotes) for meta expressions.

4.1 Exploring programmer language

Delorey et al. [9] note a series of developments that conspire to make it reasonable to apply linguistic techniques to study programs: 1) the emergence of *corpus linguistics* which emphasizes the study of *language use* based on data from *written language*, 2) the increase in publicly available source code due to the open source software movement, and 3) the advent of the software repository mining community. The authors propose expanding the study of programming language to include not only its design and theory but also its use by practitioners, which is in agreement with our claims in Section 2.1.

Baniassad and Myers [2] argue that a program can be seen as both the definition and sole usage of a program-specific language. In such a view, emphasis is put on the language-constructing activities of creating and naming abstractions. Indeed, the naming of abstractions is simultaneously both how the basic language is extended to suit the domain of the program, and the very means by which the program is constructed. However, this view does not take into account *intertextuality* [21] between programs written in the same programming language, and indeed between programs written in similar programming languages. This is what allows a Java programmer to make sense out of both Java programs written by other programmers and even programs written in Python or C++. If we see programs as separate languages, we fail to recognise the influence of others programs on our program. We argue that the

meaning of identifiers does not stem from the implementations provided in our program alone, but also from tradition and experience.

Liblit et al. [26] find that there is rhyme and reason to the way abstractions are created and named. Investigating the cognitive role played by names in programs, the authors find that names are far from arbitrary. Rather, programmers select and use names in cognitively motivated ways. Indeed, there are underlying principles for how abstractions are created and named. These principles are shared among programmers. Inspired by Lakoff and Johnson [22], the authors identify common *metaphors* employed by programmers, such as Methods are Actions and Methods are Mathematical Functions. The idea of conceptual metaphors shared by programmers is also investigated by Blackwell [4]. By analysing the JavaDoc of a corpus of Java libraries, the author finds a rich set of textual metaphors used to describe the relationships between program entities. In this case, the analysed texts are not programs per se (although they are technically embedded in the programs), but rather texts about programs. Still, these texts are clearly related to the text in the programs, and the metaphors are interesting because they relate to the elements of the programs (for instance, Components are Agents of Action in a Causal Universe). Milner [28] makes the interesting claim that the metaphor of a *listener* in event-driven programming in Java is broken, leading to much confusion among novice programmers. According to Milner, the metaphor is broken since implementation-wise, no-one is really listening. Hence confusion arises from a discrepancy between metaphor and implementation — a "naming bug" (Chapter 9), as it were. A more appropriate metaphor for event-driven programming is *subscription*.

According to Biggerstaff [3], the mapping between human-oriented concepts and implementation-oriented concepts is of critical importance for program understanding. The author refers to this mapping as *the concept assignment problem*. A program is written to accomplish some goal articulated in high-level human concepts. However, the program accomplishes this goal indirectly — the written program is a projection into a space of implementation concepts. The concepts used in human descriptions of the purpose and structure of a program are qualitatively different from the concepts used in the implementation. When reading and attempting to understand a program, the reader must mentally reverse engineer the process of translating between the concepts. The names of types and methods can be seen as living in the mapping space between human-oriented and implementation-oriented concepts. This means that high-quality names provide crucial support for the task of program understanding.

Deißenböck and Pizka [8] address the notion of identifier quality as something entirely subjective and relativistic. The authors present a formal model based on bijective mappings between concepts and names. The model is used to derive precise rules for what constitutes *concise* and *consistent* naming. To be concise, a name should hit the right abstraction level for the concept it represents: in other words, it should not be so general as to be rendered meaningless, as noted by Blackwell [5]. To be consistent, there should be a one-to-one relationship between names and concepts — in other words, no synonyms or homonyms. The authors make no distinction between kinds of concepts: these could be either domain-specific concepts or generic programming concepts. A human expert is required to provide and keep up-to-date the mapping between names and concepts. Recognising this requirement as cumbersome, Lawrie et al. [23] propose to derive syntax-based rules for identifier conciseness and consistency instead. This yields less precise results but nevertheless identifies genuine rule violations. Lawrie et

al. have also investigated ways of quantifying identifier quality [24] and the impact of identifier abbreviations on program comprehension [25].

Caprile and Tonella [6] investigate the rules of function identifier structure in C programs. As in Java method names, function identifiers in C tend to be composed of several words. By means of iterative refinement, the authors arrive at a regular grammar for the composition of words in function identifiers. The grammar goes beyond simple part-of-speech tags, providing instead classifications of the *roles* played by each word. In addition, the authors provide a concept lattice derived from the individual identifiers, highlighting both important concepts and how they relate to one another.

4.2 Finding meaningful artefacts in programs

Much research has been devoted to analysing program code with the intent of extracting artefacts that are meaningful. The goals are typically related to the intangible and fleeting property of *quality*, which we clearly want our programs to inhibit. To aid in the quest for quality, researchers have come up with strategies for identifying partial structures that are *good* (for instance design patterns), *bad* (for instance code clones) or *educational* (for instance code examples).

4.2.1 Finding patterns

A *design pattern* is a well-understood and documented solution to a common programming problem in a given context. The presence or absence of design patterns is sometimes taken as an indicator of program quality. This view is understandable, albeit a bit naive, as we have no guarantee that the right pattern has been applied to the right problem in the right context. Indeed, the seminal book on design patterns [13] emphasises this point strongly, noting that the application of design patterns always involves judgement and trade-offs. Nevertheless, researchers have been fervently investigating ways of using data mining to identify design patterns in code, arguing that it might aid program understanding and maintenance. Dong et al. [11] present a review of more than twenty approaches, classifying each with respect to such criteria as automation degree and matching degree. One of the approaches [17] uses a notion of *fingerprints* using external attributes such as number of methods, number of supertypes and subtypes, as well as cohesion and coupling to characterise classes.

A fundamental problem with identifying design patterns in programs is that they have no formal specifications; rather, they are blueprints for implementations. Gil and Maman [14] introduce the notion of *micro patterns*, which are machine-recognizable implementation patterns on the class level. In other words, a micro pattern can be expressed as a formal condition on the structure of a class. In addition, the authors demand that the patterns be *purposeful*, *prevalent* and *simple*. Of particular interest to us is the notion of *purpose*, which indicates the meaning of the pattern — why it is applied. The purpose is summarized in a *name* for the pattern. Interestingly, Singer and Kirkham [36] investigate the correspondence between micro patterns found in Java code with the *suffixes* of the corresponding class names. They find some support for their hypothesis, that the class name suffix is often an indicator of micro patterns exhibited by that class.

4.2.2 Finding clones

A *code clone* is a set of two or more program structure fragments that are essentially the same. Code clones are generally seen as the result of bad coding habits. The typical example is clones that are due to "cut-and-paste" of source code from one location in a program to another. This practice violates well-known engineering rules of thumb in the industry, affectionately known as the DRY[1] or SPOT[2] principle. The reason is that code clones can easily lead to maintenance problems: a bug must be fixed in all instances of a clone, for example.

Researches have suggested a wide array of approaches to identify potential code clones in programs. Roy et al. [32] present a comprehensive comparison and evaluation of techniques and tools. The approaches are grouped into textual, lexical, syntactical and semantic approaches. Semantics-aware approaches employ static analysis to provide more precise information than simple syntactic similarity. This involves abstracting over expressions and statements that are semantically equivalent and searching for isomorphic subgraphs in the resulting program graph.

4.2.3 Finding examples

Programs are rarely written as self-contained, isolated islands. Rather, programmers rely on existing libraries and frameworks to provide functionality from which they can compose their programs. Libraries and frameworks expose their functionality by means of application programming interfaces (APIs). An API presents the programmer with a learning curve to overcome: the programmer must somehow figure out how to use the API to accomplish the task at hand. This process can be greatly simplified if relevant code examples are present. Arguably, the best source for such examples is in programs that already successfully use the API to accomplish similar tasks. Researchers have therefor been investigating ways in which to search for relevant examples in program repositories.

Searching for artefacts can be done in many ways. A basic distinction is between characterising the artefact to look for itself, and characterising the context in which it is used. An example of the former is Sourcerer, presented by Bajracharya et al. [1]. Sourcerer is a source code search engine that uses a notion of *fingerprints* to identify artefacs. The fingerprints can be either control structure fingerprints, Java type fingerprints or micro pattern fingerprints. In other words, Sourcerer looks for artefacts in the program structure based on an abstraction over the implementation.

By contrast, Holmes and Murphy [18] use structural context to recommend source code examples. Their tool, Strathcona, uses certain *heuristics* to match the structure of code the programmer is writing to existing structures in a repository. Again, these heuristics represent an abstraction over the implementation, to facilitate comparison of program structure. The heuristics used by Strathcona include *inheritance*, *method calls* and *type usage*. Rather similarly, Sahavechaphan and Claypool [34] search for so-called "code-snippets" using a notion of *code contexts*. When writing method m in class C, the code is simultaneously in a *parent context* (defined by the supertypes of C) and in a *type context* (defined by the types referenced in m). From these contexts, the authors derive a set of code queries, ranging from generalised to specialised. The

[1] Don't Repeat Yourself [19].
[2] Single Point Of Truth [31].

queries are then used to search for relevant code snippets.

Mandelin et al. [27] present a different take on the task of finding examples of API usage. The technique can best be described as *type chaining*. Using an API to accomplish something useful is cast as a transition from a source type to a target type. The authors introduce the notion of *jungloids*, which are mappings from one type to another. Simple jungloids can be mined from a repository of API usage examples. Jungloids are composable, however, so that a long chain of type transitions can be modelled by nesting jungloids. A programmer can use the Prospector tool to automatically synthesise code examples based on jungloids.

4.3 Relating names to meaningful artefacts

Pollock et al. [30] introduce the notion of *natural language program analysis* (NLPA), which aims to exploit natural language clues found in identifier names to inform and augment program analysis. Recognising that method identifiers tend to contain verbs designating *actions* and nouns designating *actors*, the authors build a program model called an *action-oriented identifier graph* (AOIG). This graph captures the relationships between actions and actors as found in the identifiers. Given that names are applied ad-hoc by programmers, NLPA is by nature unsound. Hence, it is most amenable to problems where a certain rate of false positives and negatives are acceptable, such as aiding in program understanding and navigation.

An interesting application of NLPA is in locating *concerns* that are semantically related through some high-level *concept*, yet physically scattered in the source code. Using NLPA, Shepherd et al. [35] have implemented Find-Concept, a semi-automated concern location and comprehension tool. Of course, scattered concerns are often discussed in the context of Aspect-Oriented Programming (AOP), which aims at improving program modularisation by providing a mechanism to avoid the scattering. Locating the concerns, then, is a prerequisite for successful application of AOP.

By itself, AOP is an example of bridging the semantic gap between the informal yet meaningful names, and the formal semantics of the program graph. AOP works by injecting a piece of code, known as an *aspect*, at certain places in the program graph, as specified by the programmer. This is known as *applying* the advice at the relevant *join points*. To specify the join points, the programmer must define a *pointcut*. In theory, a pointcut can provide an arbitrary way of selecting join points, but in practice, some abstraction over method names or signatures is used. In other words, the programmer exploits the regularity or patterns in method names to directly influence the semantics of the method. (Hence one would expect programs written using AOP to be particularly structured with respect to naming.)

Chapter 5

Contribution

Here we discuss the contribution of the thesis.

5.1 Research goals

We discuss the contribution in terms of the research goals that were introduced in Section 1.1 and elaborated upon in Section 3.4. These contributions were summarised in Section 1.2; here we provide a more detailed discussion.

5.1.1 Goal G1: Name patterns

We claim that method names are natural language expressions with grammatical structure. We see hints of this in our initial investigation in *The Programmer's Lexicon* (Chapter 7), where we find that we can associate meaning with the leading verbs in method names. *The Java Programmer's Phrase Book* (Chapter 8) contains a much more thorough treatment. In the latter paper we perform syntactic analysis of method names. The analysis includes decomposing method names into individual words, part-of-speech tagging, and developing a simple notation to express method name patterns. The notation allows for mixing concrete words, word classes and wildcards (for instance **get-[adjective]-***, which covers method names like **getLastElement**). We introduce the term *Programmer English* to refer to the special dialect of English found in Java method names. We find that method names indeed exhibit grammatical structure, but it is fairly degenerate and simple. Nearly 40% of method names have a grammatical structure matching [**verb**]-[**noun+**] (meaning a single verb followed by one or more nouns), whereas nearly 80% of method names can be accounted for with the top ten grammatical structures. Most of these structures have a leading verb. The exceptions are due to "degenerate" names such as **length** (where one can imagine an implicit leading **get**), or converters such as **to-[type]**.

5.1.2 Goal G2: Usage semantics

In Section 3.4.2, we specify the requirements for a *formal semantics* of method names. We approach the problem of comparing method implementations by radically abstracting over the method implementation. The purpose of the abstraction is to highlight the essential aspects of the method implementations and ignore superficial differences. We therefore introduce the notion of a *semantic profile*. The semantic profile for a method

m is defined in terms of a set of *attributes*. Attributes are simple logical predicates defined on Java bytecode. The profile can be represented as a bit string, corresponding to evaluating each attribute on the bytecode of m in a specific order. The semantics of a method name n is defined as the collection of the bit strings for all the methods sharing the name n. Note that n is really a name pattern, rather than an actual method name.

The essential idea of a formal semantics based on attribute-profiles is introduced in *The Programmer's Lexicon* (Chapter 7) and carried forward throughout the subsequent papers. However, since the different papers address different aspects of naming, the set of attributes varies somewhat. For the purposes of characterising names in *The Java Programmer's Phrase Book* (Chapter 8), for instance, we prefer a set of fairly "broad" attributes that focus on typical behaviour. For the purposes of discovering naming bugs in *Debugging Method Names* (Chapter 9), on the other hand, we find that "narrow" attributes are useful — in order to identify particular unsuitable or "forbidden" behaviour.

5.1.3 Goal G3: Understanding naming

In Section 3.4.3, we enumerated some key points of investigation in order to better understand the meaning of names. Here we discuss our contributions relating to each point.

Characterisation. The usage semantics gives us a characterisation of a group of method implementations, and indirectly of the method name shared by such a group. In *The Programmer's Lexicon* (Chapter 7), we exploit the characterisation to mechanically generate a textual description of methods that share the same verb. The output of the generation is a "lexicon" containing textual descriptions of each verb commonly used in Java programs. In *The Java Programmer's Phrase Book* (Chapter 8), we perform a more sophisticated grouping of methods based on *phrases*, abstract method names that may include concrete words, word types and wildcards. We then generate textual descriptions for each significant phrase in the vocabulary of Java programmers. The descriptions are collected in a "phrase book"[1]. The phrase book is organised hierarchically, since some phrases are specialisations of others.

Since the usage semantics is defined in terms of abstraction over the method implementations, the textual description of each name[2] reflects what characterises a typical implementation for a method with that name. Note that a meaningful characterisation is only possible by means of comparison and contrast: we must relate the characterisation of one group of methods to other groups of methods. Hence, we find that methods with a given name n may be, say, more inclined to fulfil some attribute a_1, and less inclined to fulfil some other attribute a_2 than average. The characterisation of n, then, stems from noting how the implementations of methods named n deviate from the average.

Suitability. In *Debugging Method Names* (Chapter 9), we investigate what it means for a name and implementation to suit each other. More precisely, we consider *ill-*

[1] The phrase book can be browsed at `http://phrasebook.nr.no`.

[2] Here *name* is understood abstractly, and may refer to a verb or a phrase.

suited implementation characteristics for methods of a given name. In other words, we focus on the *not-meaning* of names. Instead of textual descriptions, we generate *implementation rules* based on the usage semantics for each commonly used method phrase. A violation of an implementation rule is considered a *naming bug*. We use the implementation rules to find many examples of naming bugs in well-known Java applications. We present statistics showing the prevalence of naming bugs in the corpus and discuss some salient examples of naming bugs in detail. We note that resolving a naming bug requires changing either the name or the implementation. To support the former, we propose a simple approach for automatic renaming of methods.

Similarity. In *The Programmer's Lexicon* (Chapter 7), we look at similarity with the purpose of augmenting the textual description of verbs in method names. Including a list of similar words is common in natural language dictionaries; we do the same in our lexicon of verbs. In our model, each method name is associated with a bag of implementations. Intuitively, if the implementations in two bags are similar, then the corresponding verbs/phrases hold similar meaning. In *The Programmer's Lexicon*, we use overlap of implementation cliches to identify similar names.

The investigation of similarity is taken much further in *Canonical Method Names for Java* (Chapter 10), where we use similarity of implementation to identify candidates for synonym unification. In *Canonical Method Names for Java*, we measure similarity using a formula that includes nominal entropy and semantic entropy. This is a more sophisticated approach that accounts for all implementations corresponding to a name, rather than just the cliches. We use the formula to mechanically generate a graph showing synonym candidates for the most commonly used verbs in Java. We also generate a list of suggestions for canonicalisation of verbs through unsupervised synonym unification.

Precision. In *The Programmer's Lexicon* (Chapter 7), we introduce *semantic entropy* as an inverse measure of precision (since precision is inversely proportional to semantic entropy); it is also used in *The Java Programmer's Phrase Book* (Chapter 8). As mentioned in Section 3.4.3, precision in naming is really consistency of implementation. Intuitively, a wide variety of implementations leads to high entropy. We associate a number for semantic entropy with each name. Comparing entropies allows us to make relative statements about how precise a certain name is. In *The Programmer's Lexicon*, we find that **size** is the most precise common verb, whereas **load** is the least precise.

Consistency. In *Canonical Method Names for Java* (Chapter 10), we are interested in identifying methods with "unnameable semantics". To do so, we introduce *nominal entropy* as an inverse measure of consistency — a semantic object with high entropy is inconsistently named. We interpret this to mean that the object has semantics that cannot be given a reasonable name.

5.1.4 Prerequisite: Representative corpus

We gather a corpus aimed at satisfying the criteria of size, variety and recognition as outlined in Section 3.4.4. The corpus consists of 100 well-known, open-source applications, frameworks and libraries written in Java. They cover a wide range of domains,

including desktop applications, developer tools, servers, implementations of programming languages, and so forth. The corpus is pruned to ensure that each class file was only included once, using the fully qualified class name. The corpus is shared between the papers. The only exception is *Canonical Method Names for Java* (Chapter 10), where the corpus was subject to some additional filtering. The reason is that synonym identification is a delicate task, particularly sensitive to noise in the data. We therefore develop a technique to mechanically identify likely instances of code generation in a corpus of methods, and impose a limit on how many near-identical methods with identical name a single application may contribute to the corpus. We also identify and eliminate *unnameable methods* from the corpus.

5.2 Critique

During the course of working with the thesis, we identified some limitations and potential issues with our approach. These are discussed below.

5.2.1 Limitations of the usage semantics model

The usage semantics model for method names is what enables our investigation into the meaning of method names. However, it has a number of limitations.

Philosophical issues. Our pragmatic theory of meaning is inspired by Wittgenstein, who claimed that "the meaning of a word is its use in the language" [42]. In Section 3.1 we note two interpretations of the word *use*: use-as-caller and use-as-implementer. Wittgenstein's claim relates to the former interpretation, yet in this thesis we have chosen the latter. The reason is that natural language phrases in programming languages include an explicit specification of meaning — the method implementation. Transferred to a conventional natural language, we can imagine each use of a phrase followed by the statement "by which I mean...".

Violation of encapsulation. A method name is not just a mirror of the implementation, or a terse way of stating the same thing the implementation does. It represents an abstraction, and also a barrier: an encapsulation. Arguably, the reader of the name should understand *what* the method does conceptually (which task it carries out), but not *how* it does it. Defining the meaning in terms of the implementation is a glass box approach. Arguably, it violates encapsulation by revealing implementation details.

Simplistic semantics. We provide a simplistic semantic profile for methods, based on a crude abstraction over method implementations. With respect to the call graph (Section 3.2), we disregard the edges (method calls), except for some attributes that capture the existence of certain method calls. The model does not handle delegation, neither at the semantic nor at the nominal level. This is problematic, since the semantics of a method arguably subsumes the semantics of the methods it calls. A possible solution at the semantic level would be to inline methods, in particular private methods. A possible solution at the nominal level would be to incorporate a nominal profile for methods, so as to note which names tend to invoke which other names.

Subjectivity of attributes. The purpose of a semantic profile based on attributes is to highlight the essential aspects of an implementation and ignore the non-essential aspects. However, it is not obvious what the essential aspects are. We have chosen to select attributes subjectively based on our own knowledge of what is significant in implementations. This means that we may have created a flawed or imperfect abstraction of method semantics. Indeed, we would have preferred to have generated the set of attributes mechanically. For instance, we could use some metric to mechanically select the most appropriate attributes from a systematically constructed pool of attributes. The benefit of using a metric instead of relying on subjective judgement is that the process is more transparent. The choice of metric is nevertheless still largely subjective, however. Even if we were to perform a triage of metrics, we would have to make a judgement regarding which is better. We have therefore sought to provide clear rationale for our selection of attributes instead.

Ad hoc reasoning. We have arrived at a number of formulas, threshold values, statistical considerations and so forth to illuminate the aspects of naming we are interested in. Viewed in retrospect, the formal framework we have developed has a tinge of exploratory, ad hoc reasoning. It may have been preferable to use *data mining* as an alternative approach. This would have given us a sound theoretical foundation and well-known techniques to apply. For instance, we could train a "naming bug finder" (confer Chapter 9) on a pre-classified set of examples[3]. This could potentially give us a more targeted set of implementation rules for a given name (or phrase), since it would not have to be constrained to a pre-defined set of attributes. However, it is not obvious how to recast all the questions we address in this thesis as data mining problems. Moreover, data mining would not relieve us of the problem of subjectivity, as data mining typically occurs iteratively, involving many opportunities for subjective choice.

5.2.2 Limitations of the corpus

We assume that the corpus is representative of real-world Java programming; that the examples of programming practice found in the corpus represent a fair cross-section of Java programming in general. We have made efforts to make it likely for this assumption to be valid, as described in Section 5.1.4. However, there are some potential problems with gathering a software corpus, in particular with respect to which applications to include, and how to ensure that the corpus is not polluted by code generation.

Selection of applications. We have selected a large number of well-known open source Java applications for our corpus, covering a wide variety of domains. We did so to create *balance* in the corpus, so that no domain or particular style of programming would be dominant. However, the applications in the corpus vary significantly by size. It is therefore a potential problem that the largest applications, such as Eclipse and the Java SDK, will dominate.

Impurity. Code generation poses a challenge for our analysis, since it has the potential to skew the data. This could in turn lead to misrepresentation of the meaning of

[3]Yossi Gil made this suggestion at the ECOOP 2009 conference.

names. Unfortunately, we are not aware of any general solution to eradicating generated code, since it is not labelled the way the compiler labels code it synthesises during compilation. We have taken some measures to diminish the effect of code generation, but there are probably many minor instances of code generation that we have not identified. For instance, we have made no effort to discover instances of code generation that involve variation of name patterns. It may be that some newer approaches, such as using the Gini coefficient to identify generated code [40], would yield better results.

5.3 Conclusion

We believe we have found substantial support for the thesis statement. Method names are meaningful phrases in natural language, a programming-specific dialect of English we refer to as *Programmer English*. The meaning of these phrases can be approximated by inspecting how they are used in practice in real-world Java programs. This approximation in turn allows us to illuminate key aspects of naming in Java.

There is an untapped potential in exploiting the meaning of method names to provide a better programming experience. Naming has been neglected for too long — we see an opportunity for *naming-aware programming*, where the computer no longer is allowed to ignore the names we use when programming.

At the lowest level of ambition, tools should be developed to aid programmers in choosing appropriate names. Using ideas from this thesis, it would be easy to create a tool to find and rectify naming bugs, for instance. One can also imagine tools to handle automatic naming of certain cliche implementations, for instance for *getters* and *setters* — but also for *finders* and *creators*. Conversely, tools could be used to generate meaningful implementation stubs from method names.

A more ambitious idea is to design new programming languages that provide better support for naming as a means of communication. The expressiveness of names is limited due to the many roles names must play in programs [12]. Relieving names from the role of linking program elements would allow the programmer to write phrases in a more operative language. In such a language, verbs could be recognised as meaningful, and nouns be linked to types as appropriate.

Finally, one could envision names as an enabling mechanism for a language that is intrinsically self-learning — in other words, a programming language that would become aware of its own patterns and idioms, by accumulating a history of name usage. This would enable a notion of *crowd-sourced programming*, harnessing the knowledge embedded in an ever-growing body of programs to improve ease of learning, reliability and productivity when writing new programs. One might even imagine a market for free or proprietary knowledge banks for programming practice by leading authorities (that is, subscribe to the programming practice of the open-source movement, Google or Microsoft!). This could enable reuse at a deeper, more philosophical level than currently envisioned. A program would no longer exist in isolation, but be informed by the wisdom embedded in its myriad of predecessors.

Bibliography

[1] S. K. Bajracharya, T. C. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. V. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In P. L. Tarr and W. R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 681–682. ACM, 2006.

[2] E. Baniassad and C. Myers. An exploration of program as language. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, pages 547–556, New York, NY, USA, 2009. ACM.

[3] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering (ICSE 1993)*, pages 482–498, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[4] A. F. Blackwell. Metaphors we program by: Space, action and society in Java. In *Proceedings of the Psychology of Programming Interest Group Conference (PPIG 2006)*, September 2006.

[5] A. F. Blackwell, L. Church, and T. Green. The abstract is 'an enemy': Alternative perspectives to computational thinking. In *Proceedings of the Psychology of Programming Interest Group Conference (PPIG 2008)*, September 2008.

[6] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE 1999), 6-8 October 1999, Atlanta, Georgia, USA*, pages 112–122. IEEE Computer Society, 1999.

[7] A. F. Chalmers. *What is This Thing Called Science?* Open University Press, 3rd edition, 1999.

[8] F. Deißenböck and M. Pizka. Concise and consistent naming. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005)*, pages 97–106. IEEE Computer Society, 2005.

[9] D. P. Delorey, C. D. Knutson, and M. Davies. Mining programming language vocabularies from source code. In *Proceedings of the Psychology of Programming Interest Group Conference (PPIG 2009)*, June 2009.

[10] E. W. Dijkstra. On the cruelty of really teaching computing science. `http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF`, December 1988.

[11] J. Dong, Y. Zhao, and T. Peng. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(6):823–855, 2009.

[12] J. Edwards. Subtext: Uncovering the simplicity of programming. In Johnson and Gabriel [20], pages 505–518.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[14] J. Gil and I. Maman. Micro patterns in Java code. In Johnson and Gabriel [20], pages 97–116.

[15] R. L. Glass. A structure-based critique of contemporary computing research. *Journal of Systems and Software*, 28(1):3–7, 1995.

[16] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):685–746, 2001.

[17] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi. Fingerprinting design patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 172–181, Washington, DC, USA, 2004. IEEE Computer Society.

[18] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 117–125. ACM, 2005.

[19] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[20] R. E. Johnson and R. P. Gabriel, editors. *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), October 16-20, 2005, San Diego, CA, USA*. ACM, 2005.

[21] J. Kristeva. *Desire in Language: A Semiotic Approach to Literature and Art*. Columbia University Press, New York, 1980.

[22] G. Lakoff and M. Johnson. *Metaphors we Live by*. University of Chicago Press, Chicago, 1980.

[23] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), 27-29 September 2006, Philadelphia, Pennsylvania, USA*, pages 139–148. IEEE Computer Society, 2006.

[24] D. Lawrie, H. Feild, and D. Binkley. Quantifying identifier quality: An analysis of trends. *Journal of Empirical Software Engineering*, 12(4):359–388, August 2007.

[25] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? A study of identifiers. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pages 3–12. IEEE Computer Society, 2006.

[26] B. Liblit, A. Begel, and E. Sweeser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, Sussex, United Kingdom, September 2006. Psychology of Programming Interest Group.

[27] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2005)*, pages 48–61, New York, NY, USA, 2005.

[28] W. W. Milner. A broken metaphor in Java. *ACM SIGCSE Bulletin*, 41(4):76–77, 2009.

[29] A. Newell and H. A. Simon. Computer science as empirical inquiry: symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.

[30] L. L. Pollock, K. Vijay-Shanker, D. Shepherd, E. Hill, Z. P. Fry, and K. Maloor. Introducing natural language program analysis. In M. Das and D. Grossman, editors, *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007), San Diego, California, USA, June 13-14, 2007*, pages 15–16. ACM, 2007.

[31] E. S. Raymond. *The Art of UNIX Programming*. Addison-Wesley, 2003.

[32] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[33] B. Russell. *Mysticism and Logic: And Other Essays*. Cornell University Library, 2009.

[34] N. Sahavechaphan and K. Claypool. XSnippet: Mining for sample code. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006), October 22-26, 2006, Portland, Oregon, USA*, pages 413–430. ACM, 2006.

[35] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development (AOSD 2007)*, pages 212–224, New York, NY, USA, 2007. ACM.

[36] J. Singer and C. Kirkham. Exploiting the correspondence between micro patterns and class names. In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008)*, pages 67–76, Beijing, China, 2008. IEEE Computer Society.

[37] I. Solheim and K. Stølen. Technology research explained. Technical report, SINTEF, 2007.

[38] G. L. Steele Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999. Original OOPSLA 1998 keynote: `http://video.google.com/videoplay?docid=-8860158196198824415`.

[39] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9–18, 1995.

[40] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the Gini coefficient. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 179–188. IEEE, 2009.

[41] S. Wagner and F. Deißenböck. Abstractness, specificity, and complexity in software design. In *Proceedings of the 2nd international workshop on the role of abstraction in software engineering (ROA 2008)*, pages 35–42, New York, NY, USA, 2008. ACM.

[42] L. Wittgenstein. *Philosophical Investigations*. Prentice Hall, 1973.

[43] J. Worrall. Structural realism: The best of both worlds? In M. Lange, editor, *Philosophy of Science: An anthology*, pages 262–279. Wiley-Blackwell, 2006.

Part II

Research papers

Chapter 6

Overview of Research Papers

Paper 1: The Programmer's Lexicon, Vol I: The Verbs

Authors: Einar W. Høst and Bjarte M. Østvold.
Publication: In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 193–202, IEEE Computer Society, 2007.
My contribution: Approximately 90%. The paper is the result of my work under the supervision of Bjarte M. Østvold. I have done the analysis and implementation, and written practically all text.
Comment: We introduce the usage semantics for method names. For simplicity, we abstract away everything but the leading verb of the names. We generate a textual description for each verb, including salient features of a typical implementation, a measure of its precision and a list of similar verbs.

Paper 2: The Java Programmer's Phrase Book

Authors: Einar W. Høst and Bjarte M. Østvold.
Publication: In *Proceedings of the 1st International Conference on Software Language Engineering (SLE 2008)*, pages 322–341, Lecture Notes in Computer Science, Springer, 2008.
My contribution: Approximately 90%. The paper is the result of my work under the supervision of Bjarte M. Østvold. I have done the analysis and implementation, and written practically all text.
Comment: We refine our analysis of method names patterns greatly. We decompose the names into phrases and perform part-of-speech tagging on the individual words. For each commonly used method phrase, we generate a textual description.

Paper 3: Debugging Method Names

Authors: Einar W. Høst and Bjarte M. Østvold.
Publication: In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, pages 219–317, Lecture Notes in Computer Science,

Springer, 2009.
My contribution: Approximately 90%. The paper is the result of my work under the supervision of Bjarte M. Østvold. I have done the analysis and implementation, and written practically all text.
Comment: We generate implementation rules rather than descriptions. We use the rules to find mismatches between names and implementations — such mismatches are considered naming bugs. The paper was awarded one of two best paper awards at ECOOP 2009.

Paper 4: Canonical Method Names for Java

Authors: Einar W. Høst and Bjarte M. Østvold.
Publication: To appear in *Proceedings of the 3rd International Conference on Software Language Engineering (SLE 2010)*.
My contribution: Approximately 90%. The paper is the result of my work under the supervision of Bjarte M. Østvold. I have done the analysis and implementation, and written practically all text.
Comment: We use usage semantics and entropy considerations to identify candidates for synonym unification.

Chapter 7

Paper 1: The Programmer's Lexicon, Volume I: The Verbs

Method names make or break abstractions: good ones communicate the intention of the method, whereas bad ones cause confusion and frustration. The task of naming is subject to the whims and idiosyncracies of the individual since programmers have little to guide them except their personal experience. By analysing method implementations taken from a corpus of Java applications, we establish the meaning of verbs in method names based on actual use. The result is an automatically generated, domain-neutral lexicon of verbs, similar to a natural language dictionary, that represents the common usages of many programmers.

7.1 Introduction

At the heart of programming is *abstraction*, the creation and naming of a set of behaviours — an implementation — to form an aggregated behaviour that can be invoked by referring to the name. The wonderful thing about abstraction is that it scales: we can build new abstractions using those we have previously created. The crucial part of abstraction is to have the name reflect the semantics of the implementation. Failure in this regard is catastrophic, as a single faulty abstraction contaminates all abstractions built on top of it.

We conclude that the problem of *naming* is vital to the task of programming. The programmer must constantly ask of herself:

- What names should I use?
- Is this a good name for the behaviour?
- Will other programmers understand the meaning of this name?
- How can I be sure that I've used a name correctly?

If we were dealing with words in natural language, we might consult a dictionary to help answer these questions. Lacking such a tool, we turn instead to the implementation that the name is supposed to represent. Consider, for instance, the following simple Java method, where the name has been omitted:

```
Person ___() {
  return p;
}
```

In spite of the missing name, we immediately notice that it is a *getter*, that is, a method whose name should start with *get*, and which returns the value of a local field.

Adding a parameter and a loop, we get:

```
Person ___(String id) {
  for (Person p : persons) {
    if (p.getID().equals(id)) {
      return p;
    }
  }
  return null;
}
```

Now we would consider it sloppy naming, if not downright wrong, if the method name started with *get*; here we are clearly trying to *find* something.

While trivial, these examples illustrate something important: there are *clues* in the implementation that can be used to indicate whether or not a name is suitable. This leads us to formulate the problem investigated in this paper: *Can we create a semantics which captures our common interpretation of method names?*

The essence of our approach is to encode the implementation of a method by means of *semantic attributes*; properties that a given implementation may or may not possess. A simple example is whether or not the implementation contains a loop. This encoding can then be used to characterise the name of the method, by aggregating all the encodings that share the same name. The characterisation constitutes a sort of "usage semantics" for the method name. Clearly, this is radically different from the formal semantics of the program itself. In a sense, it is the difference between the semantics of the informal *programmer* language on one hand and the formal *programming* language on the other.

For simplicity, we consider only the domain-neutral, action-oriented initial part of a Java method name. Typically, this is a *verb*. Hence, if the full method name is `findPersonByID`, we abstract the name to be simply *find*. In other words, we investigate the properties of *getters*, *setters*, *finders*, *closers*, *adders* and so forth. These names indicate the basic *actions* performed by Java methods. By linking these verbs to the attributes of the method implementations, we are able to create the missing dictionary of "programmer English". We call this dictionary *The Programmer's Lexicon*.

The contributions of this paper are:

- A set of attributes, defined on Java byte code, that can be used to characterise the implementation of a method (Section 7.4).
- A definition of the *usage semantics* of a name by means of the distribution of attribute-value combinations in a corpus (Section 7.2.3), establishing a formal relationship between method names and implementations.
- A measure for the precision of a name in programmer English (Section 7.3.3), based on a notion of entropy related to the semantics.

- A technique for comparing names, based on comparing their semantics (Section 7.3.4).

We use our techniques to analyse a large software corpus (Section 7.5), and explain the results by investigating some notable examples (Section 7.6).

The contributions are manifest in *The Programmer's Lexicon*, an automatically generated description of domain-neutral verbs often used in Java programming. The lexicon can be found in the appendix.

7.2 Definitions

7.2.1 Preliminaries

An *attribute* has an attribute name a and a binary value b, that is, the value is 0 or 1. For simplicity, we consider an attribute and its name as the same. An *object* o has three features: two symbols and a set of attributes $a_1, \ldots, a_m$ with values $v_1, \ldots, v_m$. The symbols are a unique *fingerprint* u and a *name* n. We use fingerprints for technical purposes and never consider their actual values. Unique fingerprints ensure that a set made from arbitrary objects $o_1, \ldots, o_k$ always has k elements, that is, they prevent that several set elements collapse into one.

A *corpus* $\mathcal{C}$ is a set of objects with the same attributes. We sometimes leave $\mathcal{C}$ implicit when there is no risk of confusion. There are two fundamental ways of dividing a corpus into parts: group objects with the same name together, or group objects with the same attribute values together. We need both.

7.2.1.1 Objects with same name

The *n-corpus* of $\mathcal{C}$, denoted $\mathcal{C}(n)$, is the set of all objects from corpus $\mathcal{C}$ that have the same name n. The *cardinality* of name n in $\mathcal{C}$, denoted $|n|_{\mathcal{C}}$, is defined as follows:

$$|n|_{\mathcal{C}} \stackrel{\text{def}}{=} |\mathcal{C}(n)|$$

where $|\mathcal{C}(n)|$ is the cardinality of set $\mathcal{C}(n)$. The *relative frequency* of an attribute a with respect to a name n, denoted $\xi_a(n)$, is the fraction of objects in $\mathcal{C}(n)$ that has attribute a set to value 1.[1]

7.2.1.2 Objects with same attribute values

If two objects o, o' have the same values for all attributes we say that they are *attribute-value identical*, denoted $o \simeq o'$. Note that this relation ignores the fingerprint and name of an object. Using relation $\simeq$ we can divide a corpus $\mathcal{C}$ into a set of equivalence classes $\mathrm{EC}(\mathcal{C}) = [o_1]_{\mathcal{C}}, \ldots, [o_k]_{\mathcal{C}}$, where $[o]_{\mathcal{C}}$ is defined as:

$$[o]_{\mathcal{C}} \stackrel{\text{def}}{=} \{o' \in \mathcal{C} \mid o' \simeq o\}.$$

We simplify the notation to $[o]$ when there can be no confusion about the interpretation of $\mathcal{C}$.

[1] Gil and Maman call $\xi_a(n)$ the *prevalence* of 'pattern' a [5].

By definition the equivalence classes of a corpus are disjoint — each object belongs to exactly one equivalence class. The cardinality $|[o]_{\mathcal{C}}|$ is the number of distinct objects in $\mathcal{C}$ that are equivalent to o by relation $\simeq$. The sum of the cardinalities of the equivalence classes equals the cardinality of $\mathcal{C}$,

$$|[o_1]| + \cdots + |[o_k]| = |\mathcal{C}|.$$

7.2.2 Distribution and entropy

We repeat some information-theoretical concepts related to entropy [2]. Let X be a discrete random variable with alphabet χ and probability mass function $p(x) = Pr\{X = x\}, x \in \chi$. Since a) for all $i = 1, \ldots, k$ it holds that $0 \leq p(x_i) \leq 1$; and b) $\sum_{i=1}^{k} p(x_i) = 1$, we have that $p(x_1), \ldots, p(x_k)$ form a *probability distribution*.

The *Shannon entropy* H of X can then be defined as:

$$H(X) \stackrel{\text{def}}{=} -\sum_{x \in \chi} p(x) \log_2 p(x)$$

where we assume $0 \log_2 0 = 0$.

The entropy of a distribution measures the uncertainty of a random variable having that distribution. Alternatively, it measures the expected number of bits required to represent an event from the distribution.

Next we define the entropy of a corpus, and based on this, the entropy of a name. Let the probability mass function $p([o])$ of corpus $\mathcal{C}$ be defined as

$$p([o]) \stackrel{\text{def}}{=} \frac{|[o]|}{|\mathcal{C}|}, \quad [o] \in \text{EC}(\mathcal{C}).$$

By the definitions of $[o]$ and cardinality, it follows that a corpus has a probability distribution $p([o_1]), \ldots, p([o_k])$. Thus we can define the entropy of corpus $\mathcal{C}$ as

$$H(\mathcal{C}) \stackrel{\text{def}}{=} H\big(p([o_1]), \ldots, p([o_k])\big).$$

Since $\mathcal{C}(n)$ also has a probability distribution, we have that a name has an entropy, denoted $H(n)$, defined as the entropy of $\mathcal{C}(n)$,

$$H(n) \stackrel{\text{def}}{=} H\big(\mathcal{C}(n)\big).$$

7.2.3 The Usage Semantics of Names

We define the *usage semantics* of a name n, written $[\![n]\!]$, in terms of $\mathcal{C}(n)$ as follows:

$$[\![n]\!] \stackrel{\text{def}}{=} \big\{([o], |[o]|) \;\big|\; [o] \in \text{EC}(\mathcal{C}(n))\big\},$$

where $([o], |[o]|)$ is the pair consisting of the equivalence class $[o]$ and the cardinality of that class.

Thus $[\![n]\!]$ reflects all the ways in which n is used in $\mathcal{C}(n)$, as well as the number of times it is used in each way. We can visualise $[\![n]\!]$ by drawing a vertical bar for each equivalence class $[o]$ in the probability distribution of $\mathcal{C}(n)$. We refer to this visualisation as the *distribution diagram* for n (see Section 7.6).

Finally, we define a function S that yields a set of equivalence classes which each cover at least a fraction q of the objects in $\mathcal{C}(n)$:

$$S(n,q) \stackrel{\text{def}}{=} \left\{([o],|[o]|) \;\middle|\; ([o],|[o]|) \in [\![n]\!] \wedge p([o]) \geq q\right\}$$

We call this a *spike set*. It has a straightforward interpretation in light of distribution diagrams, in that the most prominent equivalence classes reveal themselves as spikes in the diagrams.

7.3 Approach to Name Analysis

The names we consider in this paper are abstractions of the real method names used in Java programs. The aim is to capture the essence of the common names — typically verbs — used to denote the *actions* performed by Java methods. For instance, the concrete method names `open`, `openConnection` and `openFile` will all be considered instances of the abstract name *open*. Hence a *name* is an abstraction that will typically represent many concrete methods.

We investigate the name abstraction by looking at what is being abstracted; that is, we distil information from analysing the implementation of each method. In doing so, we apply a corpus-based *usage semantics* for names, in that the meaning is determined by the actual use of the name in a large software corpus. This is similar to how the semantics of words in natural language is established.

Of particular interest to us is the *precision* of the name, that is, how clearly the abstraction indicates the semantic content of the method, a *description* of the name, that is, what the typical semantic content is, and a *comparison* of the name to other names, in particular to find names that are similar or related in some way.

A problem not addressed in our current work is that of *polysemy*: the same name may have more than one meaning. If present, polysemy will manifest itself indirectly as lowered precision in the characterisation of the name, as well as a potentially skewed description of the name itself.

7.3.1 Restricting the Set of Names

Since the set of names used in programming is potentially unbounded, we device an algorithm for establishing the set of *common names* based on all the names in the corpus.

To ameliorate the effect of any idiosyncracies in large software projects (for instance, Sun's Java API), we sort the corpus of applications alphabetically, mechanically divide it into k subcorpora, and choose the n most frequent names in each subcorpus. Constructing the intersection of the k sets yields a set of N names, where $|N| \leq n$. This is the set of *common names*. The set of objects with common names is denoted $\mathcal{C}_{com}$. We use this set as the data material from which we establish semantic similarities and dissimilarities.

For the sake of brevity, we focus our investigation on a subset of the $m < |N|$ most frequent among the common names. This is the set of names presented in *The Programmer's Lexicon*. We write $\mathcal{C}_{lex}$ for the corresponding corpus of objects.

Percentile	*Group name*
$< 5\%$	Low extreme
$< 25\%$	Low
25% - 75%	Unlabelled
$> 75\%$	High
$> 95\%$	High extreme

Table 7.1: Quantile groups for attribute values.

The concrete values used in our analysis are $k = 5$ subcorpora with $n = 150$ candidate names from each subcorpus, yielding $|N| = 100$ common names. The number of names in the lexicon is restricted to $m = 40$.

7.3.2 Describing Names

As is the case for natural language, it makes little sense to describe a name in isolation; a symbol requires the contrast of other symbols to become meaningful. We therefore wish to say that the relative frequency of an attribute on a name, $\xi_a(n)$, is high or low compared to that of all other names.

For a given attribute a, the relative frequencies $\xi_a(n)_i$ for all names $n_i \in N$ are distributed within the boundaries $0 \leq \xi_a(n)_i \leq 1$. We divide this distribution into five named groups, based on the 5%, 25%, 75% and 95% percentiles of relative frequencies, as shown in Table 7.1. Each name then becomes associated with a certain group for a, depending on the value for $\xi_a(n)$: the 5% of names with the lowest relative frequencies end up in the "low extreme" group, and so forth.

Taken together, the group memberships for attributes $a_i, \ldots, a_k$ becomes an abstract characterisation of a name, which can be used to generate a description of it.

7.3.3 Measuring the Precision of Names

Intuitively, precision denotes how consistently a name refers to the same *thing* or combination of things. In our context, this translates to *attribute value combinations*. If a name n tends to indicate the same combinations of values for the objects in $\mathcal{C}(n)$, we think of it as precise. In other words, the more dependent the attributes are on each other for a name n, the more precise n is.

Since entropy is a measure of how independent the attributes are, we can use entropy to measure the precision of each name. A precise name has a low degree of entropy, an imprecise name a high degree. However, *low* and *high* are relative notions; hence, a name can only be precise or imprecise compared to other names. We therefore base our characterisation on quartiles: the names with entropy in the lowest quartile are deemed *precise*, in the highest quartile *imprecise*.

7.3.4 Comparing and Relating Names

A basic assumption for our work is that no name is completely arbitrary or imprecise. For any name, then, some equivalence classes will consist of more objects than others. These equivalence classes can be thought of as the distinguishing traits of the

Name	*Formal definition*
Returns void	The *return descriptor* is ***V***.
No parameters	The list of *parameter descriptors* is empty.
Field reader	*GETFIELD* or *GETSTATIC* instruction.
Field writer	*PUTFIELD* or *PUTSTATIC* instruction.
Contains loop	Jump instructions that allow for instructions to be executed more than once in the same method invocation.
Creates object	*NEW* instruction.
Throws exception	*ATHROW* instruction.
Type manipulator	*INSTANCEOF* or *CHECKCAST* instruction.
Local assignment	One of the *STORE* instructions (for instance, *ISTORE*).
Same name call	Calls a method of the same name.

Table 7.2: The attribute catalogue.

name. We exploit this fact to compare individual names, with the aim of characterising the relationship between them. This allows us to conveniently ignore inevitable variations in precision and nuance between names, and focus on the essential similarities or differences.

We use the *spike sets* $S(n_1, q)$ and $S(n_2, q)$ (see Section 7.2.3) to characterise two names n_1 and n_2 as being:

- *Similar*, in which case $S(n_1, q) = S(n_2, q)$.
- *Generalisations* or *specialisations* of each other. We say that n_1 generalises n_2 (and, conversely, that n_2 specialises n_1) if $S(n_1, q) \subset S(n_2, q)$.
- *Somewhat related*, when $S(n_1, q) \cap S(n_2, q) \neq \emptyset$.

The value for q must be set based on human judgement — we simply choose the value that seems to yield the best results: $q = 0.1$.

7.4 The Attribute Catalogue

Gil and Maman [5] define the term *traceable pattern* as "a simple formal condition on the attributes, types, name and body of a software module and its components." Here formal means that a program can check if a module matches a pattern or not. The term module includes packages, classes, methods, procedures, and fragments of code, code attributes or names. Design patterns [4] are not traceable: they cannot be recognised mechanically. A traceable pattern on a method or procedure is called a *nano pattern*.

We do not propose nano patterns here; rather, we define a set of *traceable attributes* that could be used as building blocks for creating such patterns. An attribute is *traceable* if its value can be determined mechanically. We also require that the attributes be *independent*, in the sense that the value of an attribute cannot be derived logically from another.

We define our attributes in terms of formal conditions on the byte code. We have chosen to analyse byte code because it is easily available both for open source and

commercial applications, and because we are then guaranteed to analyse the actual code that runs.

The attributes are listed in Table 7.2. For explanations of the terms used in the formal definitions, see Lindholm and Yellin [8]. The selection is based on our experience as Java programmers. The attributes are meant to indicate the basic, generic behaviours of a method implementation. For instance, *field writer* indicates that the method alters the state of an object, *same name call* hints at recursion or delegation, and so forth.

7.4.1 Critique of the Catalogue

Our current choice of attributes is somewhat arbitrary, in the sense that it rests on our intuitions about what distinguishes methods from each other. A more structured approach would be to use the marginal entropy [5] of individual attributes to select from a pool of candidate attributes those that provide the best separation power. That way, we would rely less on our own preconceptions.

Furthermore, the quality of attributes is limited by the sophistication of our current analysis. Using simple data flow analysis [9], for instance, we could define more poignant attributes such as "return value stems from field", or "parameter value is written to field".

7.5 The Corpus of Java Programs

We introduce some informal terms to aid in the discussion of our data set. By *application* we mean a compiled Java application having an intended use. Applications may range widely in domain and complexity, from the lithe JUnit testing framework to the massive JBoss Application Server. A software *collection* is a set of applications. A *corpus* is large collection chosen deliberately to cover a spectrum of intended purposes, to ensure that it is representative of all kinds of applications.

We had two main goals when gathering applications for the software corpus: we wanted it to be as large as possible, and we wanted it to consist of applications that are commonplace or well-known.

We identified several groups of applications to help balance the corpus, and to make sure it covered a wide range of domains: desktop applications, programmer tools, languages, language tools, middleware, servers, software development kits, XML tools and common utilities. Note that this grouping was not intended to be an exhaustive taxonomy for applications, but rather to act as a skeleton to span the extent of our corpus. The resulting list of applications is presented in Table 7.3.

Since applications are rarely built from scratch, they often contain dependencies upon other bits and pieces of software, ranging from applications to libraries to individual class files. Hence, the corpus is littered with all kinds of additional applications that we did not originally plan to include.

In principle, we would like to identify, separate and label all the different applications in the corpus. In practice, this task is infeasible due to the multitude of applications and versions, and the myriad ways they can be combined and intertwined. Instead, we chose to eliminate JAR files that contained many classes that collide with classes in other JAR files, that is, when the classes had the same fully qualified name.

Applications	
Desktop applications	
ArgoUML 0.24	JEdit 4.3
Azureus 2.5.0	LimeWire 4.12.11
BlueJ 2.1.3	NetBeans 5.5
Eclipse 2.3.1	Poseidon CE 5.0.1
Programmer tools	
Ant 1.7.0	FitNesse
Cactus 1.7.2	JUnit 4.2
Cobertura 1.8	Maven 2.0.4
CruiseControl 2.6	Velocity 1.4
Languages	
BeanShell 2.0b	Jython 2.2b1
Groovy 1.0	Kawa 1.9.1
JRuby 0.9.2	Rhino 1.6r5
Language tools	
ANTLR 2.7.6	MJC 1.3.2
ASM 2.2.3	JavaCC 4.0
AspectJ 1.5.3	Polyglot 2.1.0
BCEL 5.2	
Middleware and frameworks	
AXIS 1.4	PicoContainer 1.3
Jini 2.1	Spring 2.0.2
JXTA 2.4.1	Struts 2.0.1
OpenJMS 0.7.7a	Tapestry 4.0.2
Mule 1.3.3	
Servers	
Geronimo 1.1.1	Jetty 6.1.1
James 2.3.0	JOnAS 4.8.4
JBoss 4.0.5	Tomcat 6.0.7b
Software development kits	
Google Web Toolkit 1.3.3	Java 6 SDK
Java 5 EE SDK	Sun Wireless Toolkit 2.5
XML tools	
Castor 1.1	Xerces-J 2.9.0
JDOM 1.0	XOM 1.1
Saxon 8.8	
Common utilities	
Hibernate 3.2.1	Log4J 1.2.14

Table 7.3: Original list of corpus applications.

The pruned corpus contains:

- 1004 JAR files
- 190572 class files
- 1384205 non-constructor methods
- 157779 omitted methods
- 1226426 included methods

We enforce rather strict qualifications for the methods to be included in the corpus. In addition to ignoring constructors, we also omit all synthetic methods. Furthermore, we demand that method names follow the standard camel-case convention for Java, use letters or digits only, and consist of more than a single character. For instance, the method name `getParser()` is included, whereas `get_parser()`, `getParser$1()` and `f()` are all omitted. The primary rationale for this strictness is that the camel-case convention is so well-established and well-known that we consider it a sign of noise when it is not followed. For instance, it might indicate that the code was generated.

7.6 Experimental Results

We perform a fully automated analysis of the software corpus. The output of our analysis is summarised in *The Programmer's Lexicon*, printed in the appendix.

As an example illustrating both how the lexicon is constructed and how to read it, we look at the name *get* and its closest neighbours semantically. Note that the observations we make merely mimic those made mechanically by our analysis software. The name *get* is interesting because it is by far the most common one; nearly a third of all Java methods in the corpus are *get*-methods.

The Programmer's Lexicon defines *get* as follows:

> **get.** The most common method name. Methods named *get* often read state and have no parameters, and rarely return void, call methods of the same name, manipulate state, use local variables or contain loops. A similar name is *has*. Specialisations of *get* are *is* and *size*. A somewhat related name is *hash*.

A Java programmer should not be very surprised by this description: *get* methods tend to be short and simple functions that read object state. That methods starting with the name *has*, *is*, *size* and *hash* fit more or less the same description also matches intuition.

The entry for each name is generated by combining several pieces of information; the frequency and entropy of the name, an account of how its usage semantics compare to that of other names, and the spikes showing the most common attribute combinations for the name.

The description of the characteristics of methods with a given name is based on how the relative frequencies of attributes compare to methods with other names in the same corpus. For instance, *get* has a relative frequency of approximately 0.694 for the *no parameters* attribute. The distribution plot in Figure 7.1 shows how this compares to other names. Each dot represents one of the *common names*. We can see that 0.694

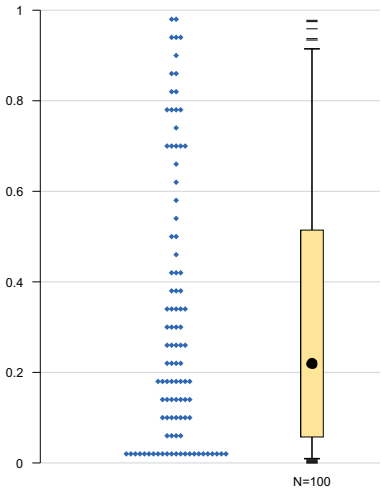


Figure 7.1: Distribution of relative frequencies for the *no parameters* attribute.

places *get* between the 75% and 95% percentiles, which leads us to characterise its score as *high*, see Table 7.1. For a mapping between quantile groups and the words used in the lexicon, see Table 7.5. It turns out that *get* has no attribute frequencies in the extremal groups.

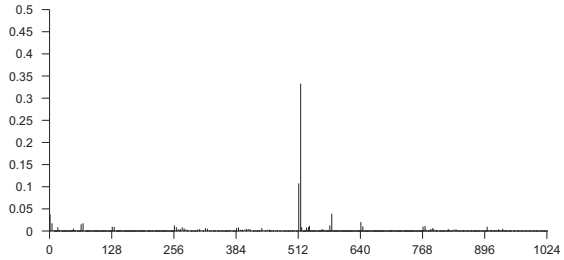
Some significant entropy values are listed in Table 7.4. We see that *get* has a higher degree of entropy than we might have anticipated. This implies that *get*-methods are not always simple field-retriever functions. Investigating the table a bit further, we notice that there are names such as *load*, with greater entropy than the corpus as a whole. The reason is that the entropy of the corpus is dominated by the most common names; again, nearly a third of all methods are *getters*. Apart from that, the most surprising entry is that of *parse*, which appears to be much more precise than we would have guessed. The explanation is that the Apache XmlBeans project, which is distributed as part of Geronimo, contributes more than 3000 near-identical *parse* methods. Presumably these have been generated. Unfortunately, there is no simple way to automatically discover generated code.

For each name n, we visualise the probability distribution for $\mathcal{C}(n)$ by means of a distribution diagram. The height of each vertical bar is $p([o])$, meaning that the y-axis signifies the fraction of objects belonging to an equivalence class. Since we use ten binary attributes, we have 2^{10} equivalence classes, yielding a resolution of 1024 on the x-axis.

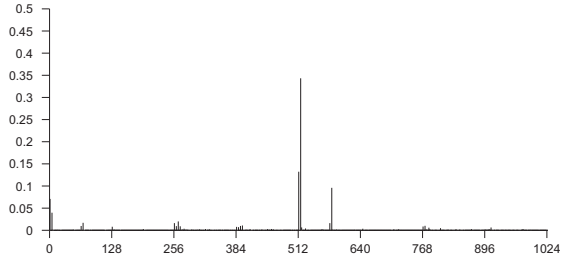
Figure 7.2 shows the distributions for *get* and its semantic neighbours[2]. For clarity, the relationship between the names is also illustrated in Figure 7.3, where similar names are connected with a bold line, specialisations point to generalisations, and somewhat related names are connected with a dotted line.

Recall that an equivalence class is included in the spike set $S(n, q)$ for a name n if

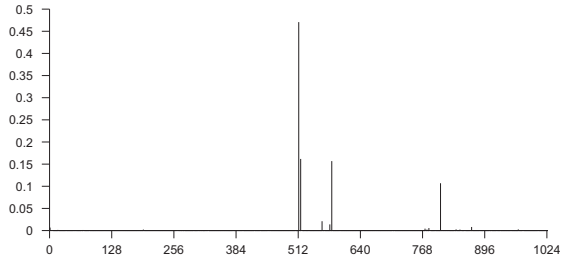
[2]Except *is*, which is omitted because it resembles *size*.



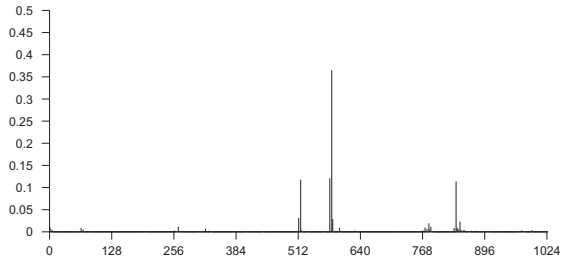
(a) *get*



(b) *has*



(c) *size*



(d) *hash*

Figure 7.2: Distribution of *get* and its semantic neighbours.

Corpus	*Entropy*	Comment
$\mathcal{C}$	6.8893	All names
$\mathcal{C}_{com}$	6.7591	Common names
$\mathcal{C}_{lex}$	6.5931	Lexicon names
$\mathcal{C}(size)$	2.5343	Most precise name
$\mathcal{C}(load)$	7.4798	Least precise name
$\mathcal{C}(get)$	4.9966	Most common name
$\mathcal{C}(hash)$	3.5616	
$\mathcal{C}(has)$	4.1766	
$\mathcal{C}(is)$	4.2318	
$\mathcal{C}(parse)$	3.6886	Suspiciously low

Table 7.4: Significant entropy values.

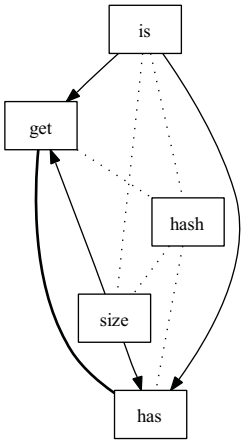


Figure 7.3: The relationships between *get* and associated names.

it accounts for at least a fraction $q = 0.1$ of the objects in $\mathcal{C}(n)$. As we can see from Figures 7.2a and 7.2b, the spikes of *get* and *has* that are higher than 0.1 correspond perfectly. This leads us to label the names as "similar".

Figure 7.2c, on the other hand, reveals that *size* has a spike that *get* does not, but not vice versa. The spike indicates a significant, specialised use; hence *size* is a specialisation of *get*. The additional spike for *size*, at position $x = 580$, represents a group of methods that read state, have no parameters, and also call other methods named *size*. This matches our intuition of what a *size* method might look like, and it also makes sense that *get* methods are not like that.

Finally, the lexicon entry says that *get* is somewhat related to *hash*. This stems from the fact that they have a spike in common (where only the attributes *reads state* and *no parameters* are set), but also that they both have spikes that are not shared by the other. To see this requires a little scrutiny of Figures 7.2a and 7.2d. The bar at position $x = 512$ for *get* represents 10.8% of all *getters*, whereas the corresponding bar for *hash* represents merely 3.1% of *hash* methods. It is more obvious that *hash* has at least one spike not shared by *get*; namely, a spike at position $x = 580$ similar to the one that differentiates *size* from *get*.

7.6.1 Exploring Nuances with a Larger Lexicon

The Programmer's Lexicon has been kept tiny for the sake of brevity and readability. Our approach can easily be used to generate a much larger and more detailed lexicon for the same corpus, allowing us to investigate more subtle nuances between names. The only prerequisite is that the cardinality for each name, $|n|_{\mathcal{C}}$, must be large enough for the analysis to be meaningful.

In such a case a printed lexicon might become unwieldy, but relationships can still be investigated meaningfully using graphs. An example graph of *dispose* and related words, taken from a lexicon generated with $n = 200$ candidate names and $m = 100$ chosen names (see Section 7.3.1), is shown in Figure 7.4. The corpus is the same that was used to generate *The Programmer's Lexicon*.

7.7 Related Work

The importance of names is well-understood by industry practitioners. In blogs and articles, fairly sophisticated discussions of names are carried out for instance by Martin Fowler, investigating the confusion caused by homonyms in source code[3], and Steve Yegge, complaining about the emphasis put on nouns over verbs in Java[4].

Among researchers, names have primarily been analysed in the context of readability and program comprehension. Deißenböck and Pizka [3] define precise rules for the conciseness and consistency of names based on a manually constructed formal model. Lawrie *et al.* [7] try to approximate the results achieved by Deißenböck and Pizka while avoiding the need for an expert to create the formal model by considering only the syntactic structure of identifiers.

A more thorough analysis of function identifiers is carried out by Caprile and Tonella [1] who investigate the structure of function identifiers in C programs, build

[3] `http://martinfowler.com/bliki/TypeInstanceHomonym.html`

[4] `http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html`

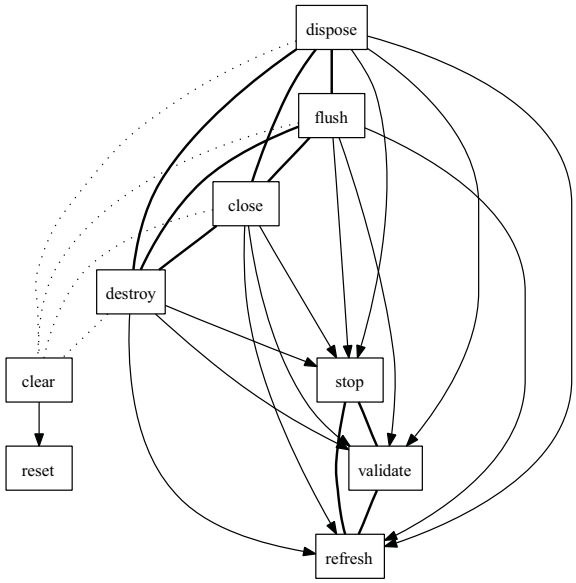


Figure 7.4: Methods related to termination.

a dictionary of identifier fragments ("words"), and propose a grammar that describes the roles of the fragments. The authors also apply concept analysis to perform a classification of the words in the dictionary.

Our work departs significantly from other efforts in the attempt to ground the *semantics* of names in attributes derived from the implementation. In trying to define traceable attributes to correlate to names, we have been influenced by Gil and Maman's work on Micro Patterns [5].

7.8 Conclusion

We believe that the analysis of semantic relationships between names in computer programs bears many low-hanging fruits, and that in generating *The Programmer's Lexicon*, we have picked but one.

Defining the meaning of names is useful because it leads to greater awareness and might contribute to more precise use of names. It is easy to envisage a tool, for instance an Eclipse plug-in, that could automatically check whether or not the initial verb of a method name suits the implementation of the method, give warnings when imprecise names are used, and so forth.

More radically, we could detach the action-oriented verbs from the rest of the method names, and raise the verbs to the status of syntax in the language. Then the programmer could write something like `Person find PersonByID`, and have the compiler verify that the implementation is not in conflict with the action specified by the name *find*.

The idea outlined above marks the beginning of a decomposition of the method

name into a more operative language, similar to *keyword messages* in Smalltalk [6]. In our present work, we have focused on the verbs that tend to form the beginning of a method name. As Caprile and Tonella [1] have shown, these verbs form part of a structure; a sentence. In the example above, if we were to identify `By` as a special word, the logical next step would be to try to link the identifier fragments `Person` and `ID` to types. The goal would be to generate a full lexicon for all the words that appear in method names, and potentially type names as well.

Since methods tend to invoke other methods, understanding the content of a method body is inherently a recursive problem. In our future work, a key challenge will be to define a suitable model that allows us to define names in terms of other names, much as is the case in natural language.

Acknowledgements

We thank Anders Moen Hagalisletto, Thor Kristoffersen and Gerardo Schneider for comments on earlier drafts of this paper.

Bibliography

[1] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE 1999), 6-8 October 1999, Atlanta, Georgia, USA*, pages 112–122. IEEE Computer Society, 1999.

[2] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. Wiley, 2nd edition, 2006.

[3] F. Deißenböck and M. Pizka. Concise and consistent naming. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005)*, pages 97–106. IEEE Computer Society, 2005.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[5] J. Gil and I. Maman. Micro patterns in Java code. In R. E. Johnson and R. P. Gabriel, editors, *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), October 16-20, 2005, San Diego, CA, USA*, pages 97–116. ACM, 2005.

[6] W. LaLonde. I can read C++ and Java but I can't read Smalltalk. *Journal of Object-Oriented Programming*, pages 40–45, 2000.

[7] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), 27-29 September 2006, Philadelphia, Pennsylvania, USA*, pages 139–148. IEEE Computer Society, 2006.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Prentice Hall, 2nd edition, 1999.

[9] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2006.

Phrase	*Meaning*
Always	The attribute value is always 1.
Very often	The name is in the high extreme quantile.
Often	The name is in the high quantile.
Rarely	The name is in the low quantile.
Very seldom	The name is in the low extreme quantile.
Never	The attribute value is always 0.

Table 7.5: Lexicon terminology.

7.A The Lexicon

Below we print *The Programmer's Lexicon*, automatically generated from our analysis of the most common names in the software corpus. In our context, a name is the action-oriented initial part of a Java method name; typically a verb. Like a natural language dictionary, the lexicon does not have to be read in full. Some entries we have found interesting are *check* (throws exceptions), *find* (contains loops) and *equals* (calls methods of the same name, and performs type-checking). Table 7.5 explains the terminology used in the lexicon.

Lexicon entries

accept. Methods named *accept* very seldom read state. Furthermore, they rarely throw exceptions, call methods of the same name, create objects, manipulate state, use local variables, have no parameters, perform type-checking or contain loops. The name *accept* has a precise use. A similar name is *visit*. Generalisations of *accept* are *handle* and *initialize*. Somewhat related names are *set*, *end*, *is* and *insert*.
action. Methods named *action* never call methods of the same name. Furthermore, they very often read state. Finally, they often return void, and rarely throw exceptions, have no parameters or contain loops. The name *action* has a precise use. Similar names are *remove* and *add*.
add. Among the most common method names. Methods named *add* often read state. Similar names are *remove* and *action*.
check. Methods named *check* very often throw exceptions. Furthermore, they often create objects and contain loops, and rarely call methods of the same name. Unfortunately, *check* is an imprecise name for a method.
clear. Methods named *clear* very often have no parameters. Furthermore, they often return void, call methods of the same name and manipulate state, and rarely create objects, use local variables or perform type-checking. A generalisation of *clear* is *reset*. A somewhat related name is *close*.
close. Methods named *close* often return void, call methods of the same name, manipulate state, read state and have no parameters, and rarely create objects or perform type-checking. A generalisation of *close* is *validate*. A somewhat related name is *clear*.
create. Among the most common method names. Methods named *create* very often create objects. Furthermore, they rarely call methods of the same name, read state or contain loops.
do. Methods named *do* often throw exceptions and perform type-checking, and rarely call methods of the same name. Unfortunately, *do* is an imprecise name for a method.
dump. Methods named *dump* never throw exceptions. Furthermore, they very often create objects and use local variables, and very seldom read state. Finally, they often call methods of the same name and contain loops, and rarely manipulate state. The name *dump* has a precise use.
end. Methods named *end* often return void, and rarely create objects, use local variables, read state or contain loops. Generalisations of *end* are *handle* and *initialize*. A specialisation of *end* is *insert*. Somewhat related names are *accept*, *set*, *visit* and *write*.
equals. Methods named *equals* never return void, throw exceptions, create objects, manipulate state or have no parameters. Furthermore, they very often call methods of the same name and perform type-checking. Finally, they often use local variables and read state. The name *equals* has a precise use.
find. Methods named *find* very often use local variables and contain loops. Furthermore, they often perform type-checking, and rarely return void.
generate. Methods named *generate* often create objects, use local variables and contain loops, and rarely call methods of the same name. Unfortunately, *generate* is an imprecise name for a method.
get. The most common method name. Methods named *get* often read state and have no parameters, and rarely return void, call methods of the same name, manipulate state, use local variables or contain loops. A similar name is *has*. Specialisations of *get* are *is* and *size*. A somewhat related name is *hash*.
handle. Methods named *handle* often read state, and rarely call methods of the same name. A similar name is *initialize*. Specialisations of *handle* are *accept*, *set*, *visit*, *end* and *insert*.

has. Methods named *has* often have no parameters, and rarely return void, throw exceptions, create objects, manipulate state, use local variables or perform type-checking. The name *has* has a precise use. A similar name is *get*. Specialisations of *has* are *is* and *size*. A somewhat related name is *hash*.
hash. Methods named *hash* always have no parameters, and never return void, throw exceptions, create objects or perform type-checking. Furthermore, they very often call methods of the same name. Finally, they often read state, and rarely manipulate state or use local variables. The name *hash* has a precise use. Somewhat related names are *has*, *is*, *get* and *size*.
init. Methods named *init* very often manipulate state. Furthermore, they often return void, create objects and have no parameters, and rarely call methods of the same name.
initialize. Methods named *initialize* often return void and manipulate state, and rarely call methods of the same name or read state. A similar name is *handle*. Specialisations of *initialize* are *accept*, *set*, *visit*, *end* and *insert*.
insert. Methods named *insert* often throw exceptions, and rarely create objects, read state, have no parameters or contain loops. Generalisations of *insert* are *handle*, *end* and *initialize*. Somewhat related names are *accept*, *set*, *visit* and *write*.
is. The third most common method name. Methods named *is* often have no parameters, and rarely return void, throw exceptions, call methods of the same name, create objects, manipulate state, use local variables, perform type-checking or contain loops. The name *is* has a precise use. Generalisations of *is* are *has* and *get*. Somewhat related names are *accept*, *visit*, *hash* and *size*.
load. Methods named *load* very often use local variables. Furthermore, they often throw exceptions, create objects, manipulate state, perform type-checking and contain loops. Unfortunately, *load* is an imprecise name for a method.
make. Methods named *make* very often create objects. Furthermore, they rarely return void, throw exceptions, call methods of the same name or contain loops.
new. Methods named *new* never contain loops. Furthermore, they very seldom use local variables. Finally, they often call methods of the same name and create objects, and rarely return void, manipulate state or read state.
next. Methods named *next* very often manipulate state and read state. Furthermore, they often throw exceptions and have no parameters, and rarely return void.
parse. Among the most common method names. Methods named *parse* very often call methods of the same name, read state and perform type-checking. Furthermore, they rarely use local variables. The name *parse* has a precise use.
print. Methods named *print* often call methods of the same name and contain loops, and rarely throw exceptions or manipulate state.
process. Methods named *process* very often use local variables and contain loops. Furthermore, they often throw exceptions, create objects, read state and perform type-checking, and rarely call methods of the same name. Unfortunately, *process* is an imprecise name for a method.
read. Methods named *read* often throw exceptions, call methods of the same name, create objects, manipulate state, use local variables and contain loops. Unfortunately, *read* is an imprecise name for a method.
remove. Among the most common method names. Methods named *remove* often throw exceptions. Similar names are *add* and *action*.
reset. Methods named *reset* very often manipulate state. Furthermore, they often return void and have no parameters, and rarely create objects, use local variables or perform type-checking. A specialisation of *reset* is *clear*.
run. Among the most common method names. Methods named *run* very often read state. Furthermore, they often have no parameters, and rarely call methods of the same name.
set. The second most common method name. Methods named *set* very often manipulate state, and very seldom use local variables or read state. Furthermore, they often return void, and rarely call methods of the same name, create objects, have no parameters, perform type-checking or contain loops. The name *set* has a precise use. Generalisations of *set* are *handle* and *initialize*. Somewhat related names are *accept*, *visit*, *end* and *insert*.
size. Methods named *size* always have no parameters, and never return void, create objects, manipulate state, perform type-checking or contain loops. Furthermore, they very seldom use local variables. Finally, they rarely read state. The name *size* has a precise use. Generalisations of *size* are *has* and *get*. Somewhat related names are *is* and *hash*.
start. Methods named *start* often return void, manipulate state and read state.
to. Among the most common method names. Methods named *to* very often call methods of the same name and create objects. Furthermore, they often have no parameters, and rarely return void, throw exceptions, manipulate state or perform type-checking.
update. Methods named *update* often return void and read state.
validate. Methods named *validate* very often throw exceptions. Furthermore, they often create objects and have no parameters, and rarely manipulate state. A specialisation of *validate* is *close*.
visit. Methods named *visit* rarely throw exceptions, use local variables, read state or have no parameters. A similar name is *accept*. Generalisations of *visit* are *handle* and *initialize*. Somewhat related names are *set*, *end*, *is* and *insert*.
write. Among the most common method names. Methods named *write* often return void and call methods of the same name, and rarely have no parameters. Somewhat related names are *end* and *insert*.

Chapter 8

Paper 2: The Java Programmer's Phrase Book

Method names in Java are natural language phrases describing behaviour, encoded to make them easy for machines to parse. Programmers rely on the meaning encoded in method names to understand code. We know little about the language used in this encoding, its rules and structure, leaving the programmer without guidance in expressing her intent. Yet the meaning of the method names — or phrases — is readily available in the body of the methods they name. By correlating names and implementations, we can figure out the meaning of the original phrases, and uncover the rules of the phrase language as well. In this paper, we present an automatically generated proof-of-concept phrase book for Java, based on a large software corpus. The phrase book captures both the grammatical structure and the meaning of method phrases as commonly used by Java programmers.

8.1 Introduction

Method identifiers play three roles in most programming languages. The first is a technical one: method identifiers are *unique labels* within a class; strings of characters that act as links, allowing us to unambiguously identify a piece of code. If we want to invoke that piece of code, we refer to the label. The second role is mnemonic. While we could, in theory, choose arbitrary labels for our methods, this would be cumbersome when trying to remember the correct label for the method we want to invoke. Hence methods are typically given labels that humans can remember, leading us to refer to method identifiers also as method *names*. Finally, method identifiers play a semantic role. Not only do we want labels we can remember; we want them to express meaning or intent. This allows us to recall what the method actually does. Unfortunately, we lack an established term for this role — identifiers are not just names. Rather, they are structured expressions of intent, composed of one or more fragments. Indeed they are *method phrases*, utterances in natural language, describing the behaviour of methods.

Consider the following example, defining the Java method `findElementByID`:

```
Element findElementByID(String id) {
  for (Element e : this.elements) {
    if (e.getID().equals(id)) {
```

```
        return e;
      }
    }
    return null;
  }
```

We immediately observe that the form of the method phrase is somewhat warped and mangled due to the hostile hosting environment. Since the phrase must double as a unique label for the method, and to simplify parsing, the phrase must be represented by a continuous strings of characters. But it is nevertheless a phrase, and we have no problems identifying it as such. In a more friendly environment, we would unmangle the phrase and simply write *Find element by ID*. For instance, in the programming language Subtext[1], the roles as links and names are completely decoupled: the names are mere comments for the links. This leaves the programmer with much greater flexibility when naming methods. Edwards argues that "names are too rich in meaning to waste on talking to compilers" [6].

Looking at the implementation, we see that there are several aspects that conspire to match the expectations given us by the phrase: the method returns a reference to an object, accepts a parameter, contains a loop, and has two return points. We posit that all meaningful method phrases can similarly be described, simply by noting the distinguishing aspects of their implementations. Our goal is to automatically generate such descriptions for the most common method phrases in Java.

Since the phrase and the implementation of a method should be in harmony, we cannot arbitrarily change one without considering to change the other. We want the phrase to remain a correct abstract description of the implementation of the method, otherwise the programmer is lying! Therefore, if the implementation is changed, the phrase should ideally be changed to describe the new behaviour[2]. Conversely, if the phrase is changed, care should be taken to ensure that the implementation fulfills the promise of the new phrase. Unfortunately, programmers have no guidance besides their own intuition and experience to help make sure that their programs are truthful in this sense. In particular, programmers lack the ability to assess the quality of method phrases with regards to suitability, accuracy and consistency.

Realizing that method names are really phrases, that is, expressions in natural language, allows us some philosophical insight into the relationship between the method name and the method body or implementation. Frege distinguishes between the *sign* — name, or combination of words —, the *reference* — the object to which the sign refers — and the *sense* — our collective understanding of the reference [8]. Note that the sense is distinct from what Frege calls the *idea*, which is the individual understanding of the reference. Depending on the insight of the individual, the idea (of which there are many) may be in various degrees of harmony or conflict with the sense (of which there is only one).

In this light, the creation of a method is a way of expression where the programmer provides both the sign (the method name) and a manifestation of the idea (the implementation). The tension between the individual idea and the sense is what motivates our work: clearly it would be valuable to assist the programmer in minimizing that

[1] `http://subtextual.org`

[2] However, since phrases act as links, this is not always practical: changing phrases in public APIs may break client code.

tension. Understanding the language of method phrases is a first step towards providing non-trivial assistance to programmers in the task of naming. In the future, we will implement such assistance in a tool.

We have previously shown how to create a semantics which captures our common interpretation, or sense, of the *action verbs* in method names [10]. Building on this work, we use an augmented model for the semantics, and expand from investigating verbs to full method names, understood as natural language phrases.

The main contributions of this paper are as follows:

- A perspective on programming that treats method names formally as expressions in a restricted natural language.
- The identification of a restricted natural language, *Programmer English*, used by Java programmers when writing the names of methods (Section 8.2.1).
- An approach to encoding the semantics of methods (Section 8.3.2), expanding on our previous work.
- An algorithm for creating a relevant and useful phrase book for Java programmers (Section 8.4.2).
- A proof-of-concept phrase book for Java that shows the potential and practicality of our approach (see Section 8.5 for excerpts).

8.2 Conceptual Overview

Our goal is to describe the meaning and structure of method names as found in "the real world" of Java programming, and present the findings in a *phrase book* for programmers. Our approach is to compare method names with method implementations in a large number of Java applications. In doing so, we are inspired by Wittgenstein, who claimed that "the meaning of a word is its use in the language" [18]. In other words, the meaning of natural language expressions is established by pragmatically considering the contexts in which the expressions are used.

8.2.1 Programmer English

Method names in Java are phrases written in a natural language that closely resembles English. However, the language has important distinguishing characteristics stemming from the context in which it is used, affecting both the grammar and the vocabulary of the language.

The legacy of short names lingering from the days before support for automatic name completion still influences programmers. This results in abbreviations and degenerate names with little grammatical structure. While increasing focus on readability might have improved the situation somewhat, Java is still haunted by this culture. A recent example is the `name` method defined for *enums*, a language feature introduced in Java 5.0. A more explicit name would be `getName`.

Futhermore, the vocabulary is filled with general computing terms, technology acronyms, well-known abbreviations, generic programming terms and special object-oriented terms. In addition, the vocabulary of any given application is extended with

domain terms, similar to the use of foreign words in regular English. This vocabulary is mostly understandable to programmers, but largely incomprehensible to the English-speaking layman.

We therefore use the term *Programmer English* to refer to the special dialect of English found in Java method names. Of course, Programmer English is really *Java Programmer English*, and other "Programmer Englishes" exist which might exhibit quite different characteristics. For instance, *Haskell Programmer English* is likely to be radically different, whereas *Ruby Programmer English* probably shares some traits with Java Programmer English, and *C# Programmer English* is likely to be near-identical.

8.2.2 Requirements for The Phrase Book

The main requirements for a phrase book is that it be *relevant* and *useful*. For the phrase book to be relevant, it must stem from "the trenches" of programming. In other words, it must be based on real-world data (i.e. programs), and be representative of how typical Java programmers express themselves.

The usefulness requirement is somewhat more subtle: what does it mean to be useful? Certainly, the phrase book should have a certain amount of *content*, and yet be *wieldy*, easy to handle for the reader. Hence, we want to be able to adjust the number of phrases included in the phrase book. In addition, each phrase must be useful in itself. We propose the following three requirements to ensure the usefulness of phrases: 1) each phrase must have a description that matches actual usage, 2) each phrase must have a well-understood semantics, and 3) each phrase must be widely applicable. These are requirements for *validity*, *precision* and *ubiquity*, respectively.

Since each Java application has its own specialized vocabulary, we must be able to abstract away domain-specific words. The phrase book should therefore contain both concrete phrases such as **get-last-element** and abstract ones such as **find-[noun]**. We prefer concrete phrases since they are more directly applicable, but need abstract phrases to fill out the picture.

We also decide that the phrase book should be organized hierarchically, as a tree. That way, the phrase book directly reflects and highlights the grammatical structure of the phrases themselves. This also makes the phrase book easier to browse. The phrases should therefore be organized as refinements of each other. Since a phrase represents a set of methods, its refinements are a partitioning of the set. Note that the partitioning is syntax-driven: we cannot choose freely which methods to group together. At any given step, we can only choose refined phrases supported by the grammar implicitly defined by the corpus.

8.2.3 Approach

Figure 8.1 provides an overview of our approach. The analysis consists of two major phases: data preparation and phrase book generation.

In the preparation phase, we transform our corpus of Java applications into an idealized corpus of methods. Each Java method is subject to two parallel analyses. On one hand, we analyze the grammatical structure of the method name. This analysis involves decomposing the name into individual words and the part-of-speech tagging of those words. This allows us to abstract over the method names and create phrases consisting of both concrete words and abstract categories of words. On the other hand,

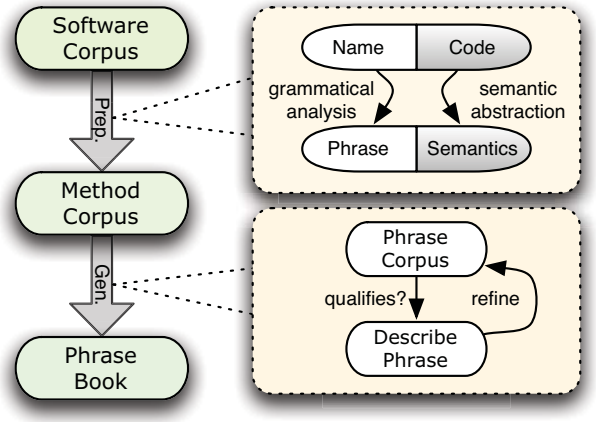


Figure 8.1: Overview of the approach.

we analyze the bytecode instructions of the implementations, and derive an abstract semantics. We can then investigate the semantics of the methods that share the same phrase, and use this to characterize the phrase.

This is exactly what happens in the generation phase. We apply our recursive phrase book generation algorithm on the corpus of methods. The algorithm works by considering the semantics of gradually more refined phrases. The semantics of a phrase is determined by the semantics of the individual methods with names that match the phrase. If a phrase is found to be useful, it receives a description in the phrase book. We generate a description by considering how the phrase semantics compares to that of others. We then attempt to refine the phrase further. When a phrase is found to be not useful, we terminate the algorithm for that branch of the tree.

8.2.4 Definitions

We need some formal definitions to be used in the analysis of methods (Sect. 8.3) and when engineering the phrase book (Sect. 8.4). First, we define a set $\mathcal{A}$ of attributes that each highlight some aspect of a method implementation. An *attribute* $a \in \mathcal{A}$ can be evaluated to a binary value $b \in \{0, 1\}$, denoting absence or presence. A *method* m has three features: a unique *fingerprint* u, a *name* n, and a list of *values* $b_1, \ldots, b_n$ for each $a \in A$. These values are the *semantics* of the method. Unique fingerprints ensure that a set made from arbitrary methods $m_1, \ldots, m_k$ always has k elements. The name n consists of one or more *fragments* f. Each fragment is annotated with a *tag* t.

A *phrase* p is an abstraction over a method name, consisting of one or more *parts*. A part may be a fragment, a tag or a special wildcard symbol. The wildcard symbol, written $*$, may only appear as the last part of a phrase. A phrase that consists solely of fragments is *concrete*; all other phrases are *abstract*. A concrete phrase, then, is the same as a method name.

A phrase *captures* a name if each individual part of the phrase captures each fragment of the name. A fragment part captures a fragment if they are equal. A tag part

captures a fragment if it is equal to the fragment's tag. A wildcard part captures any remaining fragments in a name. A concrete phrase can only capture a single name, whereas an abstract phrase can capture multiple names. For instance, the phrase **[verb]-valid-*** captures names like **is-valid**, **has-valid-signature** and so forth. The actual set of captured names is determined by the corpus.

A *corpus* $\mathcal{C}$ is a set of methods. Implicitly, $\mathcal{C}$ defines a set $\mathcal{N}$, consisting of the names of the methods $m \in \mathcal{C}$. A *name corpus* $\mathcal{C}_n$ is a set of methods with the name n. A *phrase corpus* $\mathcal{C}_p$ is a set of methods whose names are captured by the phrase p. The *relative frequency value* $\xi_a(\mathcal{C})$ for an attribute a given a corpus $\mathcal{C}$ is defined as:

$$\xi_a(\mathcal{C}) \stackrel{\text{def}}{=} \frac{\sum_{m \in \mathcal{C}} b_a(m)}{|\mathcal{C}|},$$

where $b_a(m)$ is the binary value for the attribute a of method m. The semantics of a corpus is defined as the list of frequency values for all $a \in \mathcal{A}$, $[\xi_{a_1}(\mathcal{C}), \ldots, \xi_{a_m}(\mathcal{C})]$. We write $[\![p]\!]$ for the semantics of a phrase, and define it as the semantics of the corresponding phrase corpus.

If two methods m, m' have the same values for all attributes we say that they are *attribute-value identical*, denoted $m \simeq m'$. Using relation $\simeq$ we can divide a corpus $\mathcal{C}$ into a set of equivalence classes $\text{EC}(\mathcal{C}) = [m_1]_\mathcal{C}, \ldots, [m_k]_\mathcal{C}$, where $[m]_\mathcal{C}$ is defined as:

$$[m]_\mathcal{C} \stackrel{\text{def}}{=} \{m' \in \mathcal{C} \mid m' \simeq m\}.$$

We simplify the notation to $[m]$ when there can be no confusion about the interpretation of $\mathcal{C}$. Now we apply some information-theoretical concepts related to entropy [5]. Let the probability mass function $p([m])$ of corpus $\mathcal{C}$ be defined as:

$$p([m]) \stackrel{\text{def}}{=} \frac{|[m]|}{|\mathcal{C}|}, \quad [m] \in \text{EC}(\mathcal{C}).$$

We then define the *entropy* of corpus $\mathcal{C}$ as:

$$H(\mathcal{C}) \stackrel{\text{def}}{=} H\big(p([m_1]), \ldots, p([m_k])\big).$$

Finally, we define the entropy of a phrase as the entropy of its phrase corpus.

8.3 Method Analysis

When programmers express themselves in Programmer English, they give both the actual expression (the name) and their subjective interpretation of that expression (the implementation). We therefore analyze each method in two ways: (a) a syntactic analysis concerned with interpreting the name, and (b) a semantic analysis concerned with interpreting the implementation. The input to the analyses is a Java method as found in a Java class file, the output an idealized *method* as defined in Sect. 8.2.4, consisting of fingerprint, name and semantics.

8.3.1 Syntactic Analysis of Method Names

Method names are not arbitrarily chosen; they have meaning and structure. In particular, names consist of one or more *words* (which we call fragments) put together

to form phrases in Programmer English. We apply natural language processing techniques [14], in particular *part-of-speech tagging*, to reveal the grammatical structure of these phrases.

8.3.1.1 Decomposition of Method Names.

Since whitespace is not allowed in identifiers, the Java convention is to use the transition from lower-case to upper-case letters as delimiter between fragments. For example, a programmer expressing the phrase "get last element" would encode it as `getLastElement`. Since we want to analyze the method names as natural language expressions, we reverse engineer this process to recover the original phrase. This involves *decomposing* the Java method names into fragments.

Since we are focussed on the typical way of expression in Java, we discard names that use any characters except letters and numbers. This includes names using underscore rather than case transition as a delimiter between fragments. Since less than 3% of the methods in the original corpus contain underscores, this has minimal impact on the results. That way, we avoid having to invent ad-hoc rules and heuristics such as *delimiter precedence* for handling method names with underscores *and* case transition. For instance, programmers may mix delimiters when naming test methods (e.g., `test_handlesCornerCase`), use underscores for private methods (e.g., `_findAccount`) or in other special cases (e.g., `processDUP2_X2`). Indeed, nearly half the names containing underscores have an underscore as the first character.

8.3.1.2 Part-of-speech Tagging.

The decomposed method name is fed to our *part-of-speech tagger* (POS tagger), which marks each fragment in the method name with a certain *tag*. Informally, part-of-speech tagging means identifying the correct role of a word in a phrase or sentence.

Our POS tagger for method names is made simple, as the purpose is to provide a proof-of-concept, rather than create the optimal POS tagger for method names in Java. While there exist highly accurate POS taggers for regular English, their performance on Programmer English is unknown. Manual inspection of 500 tagged names taken from a variety of grammatical structures indicates that our POS tagger has an accuracy above 97%.

The POS tagger uses a primitive tag set: **verb**, **noun**, **adjective**, **adverb**, **pronoun**, **preposition**, **conjunction**, **article**, **number**, **type** and **unknown**. Examples of **unknown** fragments are misspellings ("anonimous"), idiosyncracies ("xget") and composites not handled by our decomposer ("nocando"). Less than 2% of the fragments in the corpus are **unknown**.

A fragment with a **type** tag has been identified as the name of a Java type. We consider a Java type to be in scope for the method name if the type is used in the method signature or body. This implies that the same method name can be interpreted differently depending on context. For instance, the method name `getInputStream` will be interpreted as the two fragments **get-InputStream** if `InputStream` is a Java type in scope of the method name, and as the three fragments **get-input-stream** otherwise. As illustrated by the example, we combine fragments to match composite type names. Type name ambiguity is not a problem for us, since we only need to know that a fragment refers to a type, not which one.

Figure 8.2: Overview of the POS tagging process.

The POS tagger operates in two steps, as shown in Fig. 8.2. First, we determine the range of *possible* tags for the fragments in the phrase, then we *select* a tag for each fragment. WordNet [7] is a core component of the first task. However, since WordNet only handles the four word classes verbs, nouns, adjectives and adverbs, we augment the results with a set of prepositions, pronouns, conjunctions and articles as well. Also, since Programmer English has many technical and specialized terms not found in regular English, we have built a dictionary of such terms. Examples include "encoder" and "optimizer" (nouns) and "blit" and "refactor" (verbs). The dictionary also contains expansions for many common abbreviations, such as "abbrev" for "abbreviation".

The second step of the POS tagging is selection, which is necessary to resolve ambiguity. The tag selector is context-aware, in the sense that it takes into account a fragment's position in a phrase, as well as the possible tags of surrounding fragments. For instance, the fragment **default** is tagged as **adjective** in the phrase **get-default-value**, and as **noun** in the phrase **get-default**. Since we know that method names tend to start with verbs, a fragment is somewhat more likely to be tagged **verb** if it is the first fragment in the phrase. Also, some unlikely interpretation alternatives are omitted because they are not common in programming. For instance, we ignore the possibility of **value** being a verb.

8.3.2 Semantic Analysis of Method Implementations

The goal of the semantic analysis of the method implemenation is to derive a model of the method's behaviour. This model is an abstraction over the bytecode instructions in the implementation. In Frege's terms, we use the model to capture the programmer's idea of the method.

8.3.2.1 Attributes.

In Sect. 8.2.4, we defined the semantics of a method m as a list of binary attribute values. The attributes are predicates, formally defined as conditions on Java bytecode. We have hand-crafted the attributes to capture various aspects of the implementation. In particular, we look at *control flow*, *data flow* and *state manipulation*, as well as the *method signature*. In addition, we have created certain attributes that we believe are significant, but that fall outside these categories.

The attributes are listen in Table 8.1. Each attribute is given a name and a short description. The formal definitions of the attributes range in sophistication, from checking for presence of certain bytecode instructions, to tracing the flow of parameter and field values.

Table 8.1: Attributes.

Control Flow
Contains loop There is a control flow path that causes the same basic block to be entered more than once.
Contains branch There is at least one jump or switch instruction in the bytecode.
Multiple return points There is more than one return instruction in the bytecode.
Is recursive The method calls itself recursively.
Same name call The method calls a different method with the same name.
Throws exception The bytecode contains an `ATHROW` instruction.
Data Flow
Writes parameter value to field A parameter value may be written to a field.
Returns field value The value of a field may be used as the return value.
Returns parameter value A parameter value may be used as the return value.
Local assignment Use of local variables.
State Manipulation
Reads field The bytecode contains a `GETFIELD` or `GETSTATIC` instruction.
Writes field The bytecode contains a `PUTFIELD` or `PUTSTATIC` instruction.
Method Signature
Returns void The method has no return value.
No parameters The method has no parameters.
Is static The method is static.
Miscellaneous
Creates objects The bytecode contains a `NEW` instruction.
Run-time type check The bytecode contains a `CHECKCAST` or `INSTANCEOF` instruction.

Table 8.2: Attribute dependencies.

Contains loop $\implies$ **Contains branch**
Writes parameter value to field $\implies$ **Writes field** $\wedge$ $\neg$**No parameters**
Returns field value $\implies$ $\neg$**Returns void** $\wedge$ $\neg$**Reads field**
Returns parameter value $\implies$ $\neg$**Returns void** $\wedge$ $\neg$**No parameters**

8.3.2.2 Attribute Dependencies.

In our previous work, we used strictly orthogonal attributes [10]. However, this sometimes forces us to choose between coarse and narrow attributes. As an example, we would have to choose between common, but not so distinguishing **Reads field** attribute, and the much more precise and semantically laden **Returns field value**. We therefore allow non-orthogonal attributes in our current work.

Table 8.2 lists the dependencies between the attributes. We see that they are straight-forward to understand. For instance, it should be obvious that all methods that return a field value must (a) read a field and (b) return a value.

Note that there are more subtle interactions at work between the attributes as well. For instance, **Throws exception** tends to imply **Creates objects**, since the exception object must be created at some point. However, it is not an absolute dependency, as rethrowing an exception does not mandate creating an object.

8.3.2.3 Critique.

We have constructed the set of attributes under two constraints: our own knowledge of significant behaviour in Java methods and the relative simplicity of our program analysis. While we believe that the attributes are adequate for demonstration, we have no illusions that we have found the optimal set of attributes. A more sophisticated program analysis might allow us to define or approximate interesting attributes such as **Pure function** (signifying that the method has no side-effects). It is also not clear that attributes are the best way to model the semantics of methods — for instance, the structure of implementations is largely ignored. However, the simplicity of attributes is also the strength of the approach, in that we are able to reduce the vast space of possible implementations to a small set of values that seem to capture their essence. This is important, as it facilitates comparing and contrasting the semantics of methods described by different phrases.

8.3.3 Phrase Semantics

The semantics of a single method captures the programmer's subjective idea of what a method phrase means. When we gather many such ideas, we can approximate the sense of the phrase, that is, its objective meaning. We can group ideas by their concrete method phrases (names) such as **compare-to**, or by more abstract phrases containing tags, such as **find-[type]-by-[noun]**.

8.3.3.1 Phrase Characterization.

Just like other natural language expressions, a phrase in Programmer English is only meaningful in contrast to other phrases. The English word *light* would be hard to

Table 8.3: Percentile groups for attribute frequencies.

$< 5\%$	Low extreme
$< 25\%$	Low
25% - 75%	Unlabelled
$> 75\%$	High
$> 95\%$	High extreme

grasp without the contrast of *dark*; similarly, we understand a phrase like **get-name** by virtue that it has different semantics from its opposite **set-name**, and also from all other phrases, most of which are semantically unrelated, such as **compare-to**. Since the semantics $[\![p]\!]$ of a phrase p is defined in terms of a list of attribute frequencies (see Sect. 8.2.4), we can characterize p simply by noting how its individual frequencies deviates from the average frequencies of the same kind.

For a given attribute a, the relative frequency $\xi_a(\mathcal{C}_n)$ for all names $n \in \mathcal{N}$ lies within the boundaries $0 \leq \xi_a(\mathcal{C}_n) \leq 1$. We divide this distribution into five named groups, as shown in Table 8.3. Each name is associated with a certain group for a, depending on the value for $\xi_a(n)$: the 5% of names with the lowest relative frequencies end up in the "low extreme" group, and so forth. This is a convenient way of mapping continuous values to discrete groups, greatly simplifying comparison.

Taken together, the group memberships for attributes $a_i, \ldots, a_k$ becomes an abstract characterization of a phrase, which can be used to generate a description of it.

8.3.4 Method Delegation

The use of method delegation in Java programs — invoking other methods instead of defining the behaviour locally — is a challenge for our analysis. The reason is that a method implementation that delegates directly to another method exposes no behaviour of its own, and so it "waters down" the semantics of the method name.

There are two simple ways of handling delegation: inlining and exclusion. Inlining essentially means copying the implementation of the called method into the implementation of the calling method. There are several problems with inlining. First, it undermines the abstraction barrier between methods and violates the encapsulation of behaviour. Second, it skews the analysis by causing the same implementation to be analyzed more than once. We therefore prefer exclusion, which means that delegating methods are omitted from the analysis. However, what constitutes delegation is fuzzy: should we ignore methods that calculate parameters passed to other methods, or methods that delegate to a sequence of methods? For simplicity, we only identify and omit single, direct delegation with an equal or reduced list of parameters.

8.4 Engineering the phrase book

This section describes the engineering efforts undertaken to produce the proof-of-concept phrase book for Java programmers. In particular, we describe how we meet

the requirements outlined in Sect. 8.2.2, and take a closer look at the algorithm used to generate the phrase book.

8.4.1 Meeting the Requirements

In Sect. 8.2.2, we mandated that the phrase book be relevant and useful.

8.4.1.1 Relevance

We fulfill the relevance requirement by using a large corpus of Java applications as data for our analysis. This ensures that the results reflect actual real-world practice. The corpus is the same set of applications we used in our previous work [10]. Since many applications rely on third-party libraries, the corpus has been carefully pruned to ensure that each library is analyzed only once. This is important to avoid skewing of the results: otherwise, we cannot be certain that the semantics of a phrase reflects the cross-section of many different implementations.

The corpus consists of 100 open-source applications and libraries from a variety of domains: desktop applications, programmer tools, programming languages, language tools, middleware and frameworks, servers, software development kits, XML tools and various common utilities. Some well-known examples include the Eclipse integrated development environment, the Java software development toolkit, the Spring application framework, the JUnit testing framework, and the Azureus bittorrent client[3]. Combined, the applications in the corpus contain more than one million methods.

8.4.1.2 Usefulness

In order to produce a phrase book that is as useful as possible, we want the phrase book to be short, easy to read, and containing only the most useful phrases. Here, we present our translation of the qualitative usefulness requirements into quantitative ones.

- *Validity.* Each phrase must represent at least 100 methods.
- *Precision.* Intuitively, precision means how consistently a phrase refers to the same semantics. Since entropy measures the independence of attributes, entropy is an inverse measurement of precision. Each phrase representing a refinement of another phrase must therefore lead to *decreased* entropy, corresponding to *increased* precision.
- *Ubiquity.* Each phrase must be present in at least half of the applications in the corpus.

Tweaking the actual numbers in these criteria allows us to control the size of the phrase book. The values we have chosen yields a phrase book containing 364 phrases. The ideal size of the phrase book is a matter of taste; we opt for a relatively small one compared to natural-language dictionaries.

[3]Azureus is currently the most downloaded and actively developed application from SourceForge.net.

```
refine(phrase):
  for tag in tags:
    t-phrase = phrase-append(phrase, tag)
    if useful(t-phrase, phrase):
      used-phrases = ()
      for f-phrase in fragment-phrases(p, tag):
        if useful(f-phrase, t-phrase):
          used-phrases.add(f-phrase)
          write-entry(f-phrase)
          refine(f-phrase)
      r-phrase = mark-special(t-phrase)
      if useful(r-phrase, phrase):
        write-entry(r-phrase)
        refine(r-phrase)
```

Figure 8.3: Pseudo-code for the phrase book generation algorithm

8.4.2 Generation Algorithm

Below, we present and explain the pseudo-code (Fig. 8.3) for the algorithm that automatically generates the phrase book. Note that the pseudo-code glosses over many details to highlight the essentials of the algorithm. For brevity, we omit definitions for functions that are "self-explanatory". The syntax is influenced by Python, meaning that indentation is significant and used to group blocks.

The pseudo-code outlines a fairly simple recursive algorithm. The driving function is refine, which generates a refinement of a phrase. Note that each phrase implicitly defines a corpus of methods, so that a refinement of a phrase also means a narrowing of the corpus.

First, we iterate over the tags in our tag set (see Sect. 8.3.1.2). For each tag, we create a new phrase representing a refinement to only the methods whose names satisfy the new tag. We demand that this refinement be useful, or we ignore the entire tag. The refinement is useful if it meets the criteria of the useful function. This function embodies the criteria discussed in Sects. 8.2.2 and 8.4.1.

If the refinement is useful, we try to find even more useful refinements using fragments instead of the tag. Assume that we are calling expand on the phrase **get-***. We expand the phrase with the tag **noun**, yielding the new phrase **get-[noun]-***. Finding the new phrase to be useful, we generate more concrete refinements such as **get-name-*** and **get-customer-***. If they are useful, we call the write-entry function, which generates a description that is included in the phrase book, and recurse, by calling refine on the concrete refinement. Finally, we examine the properties of the corpus of remnant methods; those that match the tag phrase, but are not included in any of the useful concrete refinements. We say that these methods are captured by a special phrase r-phrase. The r-phrase is equal to the t-phrase, except that it potentially captures fewer method names, and hence might represent a smaller corpus. For instance, if **get-name-*** is useful and **get-customer-*** is not, then r-phrase captures the phrases captured by **get-[noun]-***, except those also captured by **get-name-***. If r-phrase is useful, it is included in the phrase book. Note that if no useful concrete refinements

Table 8.4: Phrase book terminology.

Phrase	*Meaning*
Always	The attribute value is always 1.
Very often	Frequency in the high extreme percentile group.
Often	Frequency in the high percentile group.
Rarely	Frequency in the low percentile group.
Very rarely	Frequency in the low extreme percentile group.
Never	The attribute value is always 0.

are found, r-phrase degenerates to t-phrase.

8.5 Results

While the phrase book has been designed for brevity, it is still much too large to be included in this paper. We therefore present some excerpts highlighting different aspects. The full version is available at `http://phrasebook.nr.no`. We also take a look at the distribution of grammatical structures, and using the phrase book to guide naming.

Terminology. Table 8.4 explains the basic terminology used in the phrase book. In addition, we use the modifier *comparatively* to indicate that the frequency is low despite being in the higher quantiles, or high despite being in the lower quantiles. For instance, a phrase might denote methods that call themselves recursively *more often* than average methods, even if the actual frequency might be as low as 0.1.

Example Entry. To illustrate how the data uncovered by our analysis is presented in the phrase book, we show the entry for the phrase **find-[type]**. It is an interesting example of a slightly abstract phrase with a clear meaning.

> **find-[type].** These methods very often contain loops, use local variables, have branches and have multiple return points, and often throw exceptions, do runtime type-checking or casting and are static. They rarely return void, write parameter values to fields or call themselves recursively.

Each entry in the phrase book describes what signifies the corpus of methods captured by the phrase; that is, how it differs from the average (Sect. 8.3.3.1). We find no surprises in the distinguishing features of **find-[type]**; in fact, it is a fairly accurate description of a typical implementation such as:

```
Person findPerson(String ssn) {
  Iterator itor = list.iterator();
  while (itor.hasNext()) {
    Person p = (Person) itor.next();
    if (p.getSSN().equals(ssn)) {
      return p;
```

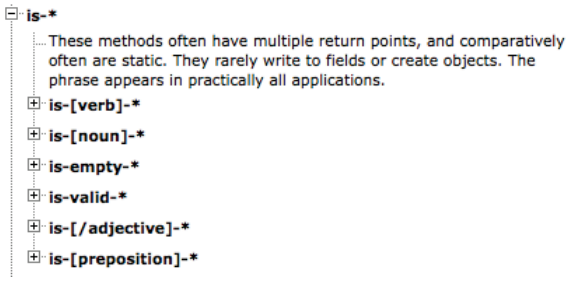


Figure 8.4: The **is-*** branch of phrases.

```
    }
  }
  return null;
}
```

We iterate over a collection of objects (**Contains loop**), cast each object to its proper type (**Run-time type check**) and store it in a variable (**Local assignment**), and terminate early if we find it (**Multiple returns**). A common variation would be to throw an exception (**Throws exception**) instead of returning `null` if the object could not be found.

Refinement. As explained in Sect. 8.4.1, the phrase book is engineered to yield useful entries, understood as valid, precise and ubiquitous ones. The generation algorithm has been designed to prefer concrete phrases over abstract ones, as long as the criteria for usefulness are fulfilled.

One effect of this strategy is that the everyday "cliché" methods `equals`, `hashCode` and `toString` defined on `Object` are not abstracted: they emerge as the concrete phrases **equals**, **hash-code** and **to-String**. This is not surprising, as the names are fixed and the semantics are well understood.

The algorithm's ability to strike the right balance between concrete and abstract phrases is further illustrated by the branch for the **is-*** phrase, shown in Fig. 8.4.

We see that the algorithm primarily generates abstract refinements for **is-***; one for each of the tags **verb**, **noun**, **adjective** and **preposition**. However, in the case of **adjective**, two concrete instances are highlighted: **is-empty-*** and **is-valid-***. This matches nicely with our intuition that these represent common method names. We write [**/adjective**] for the subsequent phrase to indicate that it captures adjectives except the preceding "empty" and "valid".

Grammar. We find that the vast majority of method phrases have quite degenerate grammatical structures. By far the most common structure is [**verb**]-[**noun**]. Furthermore, compound nouns in Programmer English, as in regular English, are created by juxtaposing nouns. The situation becomes even more extreme when we collapse these nouns into one, and introduce the tag **noun+** to represent a compound noun. The ten most common grammatical structures are listed in Table 8.5.

Table 8.5: Distribution of grammatical structures.

Structure	*Instances*	*Percent*
[**verb**]-[**noun+**]	422546	39.45%
[**verb**]	162050	15.13%
[**verb**]-[**type**]	78632	7.34%
[**verb**]-[**adjective**]-[**noun+**]	74277	6.93%
[**verb**]-[**adjective**]	28397	2.65%
[**noun+**]	26592	2.48%
[**verb**]-[**noun+**]-[**type**]	18118	1.69%
[**adjective**]-[**noun+**]	15907	1.48%
[**noun+**]-[**verb**]	14435	1.34%
[**preposition**]-[**type**]	13639	1.27%

Guidance. Perhaps the greatest promise of the phrase book is that it can be used as guidance when creating and naming new methods. Each description could be translated to a set of rules for a given phrase. An interactive tool, e.g., an Eclipse plug-in, could use these rules to give warnings when a developer breaches them.

As an example, consider the phrase **equals**, which the phrase book describes as follows:

> **equals.** These methods very often have parameters, call other methods with the same name, do runtime type-checking or casting, have branches and have multiple return points, and often use local variables. They never return field values or return parameter values, and very rarely return void, write to fields, write parameter values to fields or call themselves recursively, and rarely create objects or throw exceptions. The phrase appears in most applications.

The extreme clauses are most interesting, because they represent the clearest characteristics for the phrase. For instance, no programmer contributing to the corpus has ever let an **equals** method return a value stored in a field — a strong suggestion that you might not want to do so either! However, we note that the phrase book reflects the actual use of phrases, not the ideal use. This means that the description might capture systematic implementation problems; i.e., common malpractice for a given phrase. We might look for clues in the negative clauses, indicating rare — even suspicious — behaviour. For instance, we see that there are **equals** methods that create objects and throw exceptions, which might be considered dubious. More severely, recursion in an **equals** method sounds like a possible bug. Indeed, inspection reveals an *infinite recursive loop* bug in an **equals** method in version 1.0 of Groovy[4].

We conclude that the rules uncovered by the phrase book appear to be useful as input to a naming-assistance tool. However, the rules might need to be tightened somewhat, to compensate for fallible implementations in the corpus. After all, the corpus reflects the current state of affairs for naming, and the aim of a naming-assistance tool would be to improve it.

[4] `http://groovy.codehaus.org/`

8.6 Related Work

We build on our previous work [10], which defined semantics for *action verbs*, the initial fragment of method names. We summarized the findings in *The Programmer's Lexicon*, an automatically generated description of the most common verbs in a large corpus of Java applications. The distinguishing characteristic of our work, is that we compare the names and semantics of methods in a large corpus of Java applications.

Other researchers have analyzed Java applications in order to describe typical Java programmer practice. Collberg et al. [4] present a large set of low-level usage statistics for a huge corpus of Java programs. Examples of statistics included are the number of subclasses per class, the most common method signatures and bytecode frequencies. Baxter et al. [2] have similar goals, in using statistics to describe the anatomy of real-world Java programs. In particular, they investigate the claim that many important relationships between software artifacts follow a "power-law" distribution. However, none of these statistics are linked to names.

There have also been various kinds of investigations into identifiers, traditionally in the context of program comprehension. Lawrie et al. [12] study how the quality of identifiers affect the understanding of source code. Caprile and Tonella [3] investigate the structure of function identifiers as found in C programs. They build a dictionary of identifier fragments and propose a grammar for identifiers, but make no attempt at defining identifier semantics. Antoniol et al. [1] find that the names used in programming evolve more slowly than the structures they represent. They argue that the discrepancy is due to lack of tool support for name refactoring. In this work, names are linked to structures, but not the semantics of the structures.

Lately, more interest can be seen in investigating and exploiting name semantics. Singer et al. [17] share our ambition in ascribing semantics to names based on how they are used. They analyze a corpus of real-world Java applications, and find evidence of correlation between type name suffixes (nouns) and some of the micro patterns described by Gil and Maman [9]. Micro patterns are formal, traceable conditions defined on Java types.

Pollock et al. [15] investigate various ways of utilizing "natural language clues" to guide program analysis. Shepherd et al. [16] apply method name analysis to aid in aspect mining. In particular, they investigate the relationships between verbs (actions) and nouns (types) in programs. The scattering of the same verb throughout a program is taken as a hint of a possible cross-cutting concern. Finally, Ma et al. [13] use identifier fragments to index software repositories, to assist in querying for reusable components. These works involve exploiting implicit name semantics, in that relationships between names are taken to be meaningful. However, what the semantics are remains unknown. Our work is different, in that we want to explicitly model and describe the semantics of each name.

8.7 Conclusion

The names and implementations of methods are mutually dependent on each other. The phrase book contains descriptions that captures the objective *sense* of the phrases, that is, the common understanding among Java programmers of what the phrases mean. We arrived at the sense by correlating over a million method names and im-

plementations in a large corpus of Java applications. By using attributes, defined as predicates on Java bytecode, we modeled the semantics of individual methods. By aggregating methods by the phrases that describe them, we derived the semantics of the phrases themselves. From the semantics, we generated the textual descriptions gathered in the phrase book.

We believe that further investigation into the relationship between names and implementations can yield more valuable insight and contribute to improved naming. Currently, we are refining our model of the semantics of methods, in order to make it more sophisticated and precise. This will allow us to more accurately describe the meaning of the phrases. An obvious enhancement is to use a state-of-the-art static analysis tool to provide a richer, more descriptive set of attributes. We are also considering developing a model for the semantics that better captures the structure of the implementations. At an abstract level, it might be possible to identify machine-traceable *patterns* for method implementations. Inspired by Singer et al. [17], then, we might look for correlation between names and these patterns.

While we have shown that names do have grammatical structure, we believe that the potential for natural language expression in names is under-utilized. Indeed, by far the most common structure is the simple [**verb**]-[**noun**] structure. Longer, more complex names gives the possibility of much more precise descriptions of the behaviour of methods. Improved tool support for verifying name quality might motivate programmers to exploit this possibility to a greater extent than at the present.

We are working on transforming the results presented in this paper into a practical tool, supporting a much richer set of naming conventions than adherence to simple syntactic rules such as the camel case convention. The tool will warn against dissonance between name and implementation, and suggest two paths to resolution: 1) select a more appropriate name from a list proposed by the tool, or 2) perform one or more proposed changes to the implementation.

In a somewhat longer timeframe, the tool could be extended to support grammatical conventions as well. An example would be to warn against mixing verbose and succinct naming styles. One might debate whether it is better to explicitly mention types in method names (e.g., `Customer findCustomerByOrder(Order)`) or not (e.g., `Customer find(Order)`), but to mix both styles in the same application is definitely confusing. Tool support could help achieve grammatical consistency within the application.

Bibliography

[1] G. Antonial, Y.-G. Guéhéneuc, E. Merlo, and P. Tonella. Mining the lexicon used by programmers during sofware [sic] evolution. *Proceedings of the 23rd International Conference on Software Maintenance (ICSM 2007)*, pages 14–23, October 2007.

[2] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of Java software. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006), October 22-26, 2006, Portland, Oregon, USA*, pages 397–412. ACM, 2006.

[3] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE 1999), 6-8 October 1999, Atlanta, Georgia, USA*, pages 112–122. IEEE Computer Society, 1999.

[4] C. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. *Software Practice and Experience*, 37(6):581–641, 2007.

[5] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. Wiley, 2nd edition, 2006.

[6] J. Edwards. Subtext: Uncovering the simplicity of programming. In Johnson and Gabriel [11], pages 505–518.

[7] C. Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[8] G. Frege. On sense and reference. In P. Geach and M. Black, editors, *Translations from the Philosophical Writings of Gottlob Frege*, pages 56–78. Blackwell, 1952.

[9] J. Gil and I. Maman. Micro patterns in Java code. In Johnson and Gabriel [11], pages 97–116.

[10] E. W. Høst and B. M. Østvold. The programmer's lexicon, volume I: The verbs. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 193–202, Paris, France, 2007. IEEE Computer Society.

[11] R. E. Johnson and R. P. Gabriel, editors. *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), October 16-20, 2005, San Diego, CA, USA*. ACM, 2005.

[12] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? A study of identifiers. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pages 3–12. IEEE Computer Society, 2006.

[13] H. Ma, R. Amor, and E. D. Tempero. Indexing the Java API using source code. In *Proceedings of the 19th Australian Software Engineering Conference (ASWEC 2008), March 25-28, 2008, Perth, Australia*, pages 451–460. IEEE Computer Society, 2008.

[14] C. D. Manning and H. Schuetze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[15] L. L. Pollock, K. Vijay-Shanker, D. Shepherd, E. Hill, Z. P. Fry, and K. Maloor. Introducing natural language program analysis. In M. Das and D. Grossman, editors, *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007), San Diego, California, USA, June 13-14, 2007*, pages 15–16. ACM, 2007.

[16] D. Shepherd, L. L. Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In R. E. Filman, editor, *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006), Bonn, Germany, March 20-24, 2006*, pages 3–14. ACM, 2006.

[17] J. Singer and C. Kirkham. Exploiting the correspondence between micro patterns and class names. In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008)*, pages 67–76, Beijing, China, 2008. IEEE Computer Society.

[18] L. Wittgenstein. *Philosophical Investigations*. Prentice Hall, 1973.

Chapter 9

Paper 3: Debugging Method Names

Meaningful method names are crucial for the readability and maintainability of software. Existing naming conventions focus on syntactic details, leaving programmers with little or no support in assuring meaningful names. In this paper, we show that naming conventions can go much further: we can mechanically check whether or not a method name and implementation are likely to be good matches for each other. The vast amount of software written in Java defines an implicit convention for pairing names and implementations. We exploit this to extract rules for method names, which are used to identify "naming bugs" in well-known Java applications. We also present an approach for automatic suggestion of more suitable names in the presence of mismatch between name and implementation.

9.1 Introduction

It is well-known that maintenance costs dominate — if not the budget — then the true cost of software [7]. It is also known that code readability is a vital factor for maintenance [5]: unintelligible software is necessarily hard to modify and extend. Finally, it has been demonstrated that the quality of identifiers has a profound effect on program comprehension [14]. We conclude that identifier quality affects the cost of software! Hence, we would expect programmers to have powerful analyses and tools available to help assure that identifier quality is high.

The reality is quite different. While the importance of good names is undisputed among leading voices in the industry [2, 18, 19], the analyses and tools are lacking. Programmer guidance is limited to naming convention documents such as those provided by Sun Microsystems for Java. The following quote is typical for the kind of advice given by such documents: "Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter"[1]. In other words, the documents mandate a certain uniformity of lexical syntax. Since such uniformity is easily checked mechanically, there are tools available to check for violations against these rules. While this is certainly useful, it does little to ensure meaningful identifiers. (Arguably, syntactic uniformity helps reduce the cost of "human parsing" of identifiers, but not the interpretation.) Since identifiers clearly must be meaningful to be of high quality, current tool-support must be considered unsatisfactory.

[1] `http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html`

This begs the question what meaningful identifiers really are. Consider what an identifier is used for: it represents some program entity, and allows us to refer to that entity by means of the identifier alone. In other words, the identifier is an abstraction, and the meaning relates to the program entity it represents. The identifier is meaningful if the programmer can interpret it to gain an understanding of the program entity without looking at the entity itself. Intuitively, we also demand that the abstraction be sound: we must agree that the identifier is a suitable replacement for the entity. Hence, what we really require are identifiers that are both *meaningful* and *appropriate*.

In this work, we consider only method names. Methods are the smallest named units of aggregated behaviour in most conventional programming languages, and hence a cornerstone of abstraction. A method name is meaningful and appropriate if it adequately describes the implementation of the method. Naming is non-trivial because there is a potential for conflict between names and implementations: we might choose an inappropriate name for an implementation, or provide an inappropriate implementation for a name. The label *appropriate* is not really a binary decision: there is a sliding scale from the highly appropriate to the utterly inappropriate. Inappropriate or even meaningless identifiers are obviously bad, but subtle mistakes in naming can be as confusing or worse. Since the programmer is less likely to note the subtle mistake, a misconception of the code's behaviour can be carried for a long time.

Consider the following example, taken from AspectJ 1.5.3, where the method name has been replaced by underscores:

```
/**
 * @return field object with given name, or null
 */
public Field ___(String name) {
  for (Iterator e = this.field_vec.iterator(); e.hasNext();) {
    Field f = (Field) e.next();
    if (f.getName().equals(name))
      return f;
  }
  return null;
}
```

Most Java programmers will find it easy to come up with a name for this method: clearly, this is a *find* method! More precisely, we would probably name this method `findField`; a suitable description for a method that indeed tries to find a `Field`. The name used in AspectJ, however, is `containsField`. We consider this to be a *naming bug*, since the name indicates a question to the object warranting a boolean reply ("Do you contain a field with this name?") rather than an instruction to return an object ("Find me the field with this name!"). In this paper, we show how to derive rules for implementations of *contains* methods, *find* methods and other methods with common names, allowing us to identify this naming bug and many others. We also present an approach for automatic correction of faulty names that successfully suggests using the verb *find* rather than *contains* for the code above.

It is useful to speak of method names in slightly abstract terms; for instance, we speak of *find* methods, encompassing concrete method names like `findField` and `findElementByID`. We have previously introduced the term *method phrase* for this perspective [12]. Typically, the rules uncovered by our analysis will refer to method phrases rather than concrete method names. This is because method phrases allow

us to focus on essential similarities between method names, while ignoring arbitrary differences.

The main contributions of this paper are as follows:

- A formal definition of a naming bug (Sect. 9.3.1).
- An approach for encoding the semantics of methods (Sect. 9.3.3), building on our previous work [12, 11].
- An approach for extracting name-specific implementation rules for methods from a corpus (Sect. 9.3.4).
- An automatically generated "rule book" containing implementation rules for the most common method names used in Java programming (Sect. 9.3.4).
- An approach for automatic suggestion of a more suitable name in the case of mismatch between the name and implementation of a method (Sect. 9.3.6).

We demonstrate the usefulness of our analysis by finding genuine naming bugs in well-known Java applications (Sect. 9.5.2).

9.2 Motivation

Our goal is to exploit the vast amount of software written in Java to derive name-specific implementation rules for methods. Our approach is to compare the names and implementations of methods in a large corpus of well-known open-source Java applications. In this section, we motivate our approach, based on philosophical considerations about the meaning of natural language expressions.

9.2.1 The Java Language Game

We have previously argued that method identifiers act as hosts for expressions in a natural language we named *Programmer English* [12]. Inspired by Wittgenstein and Frege, we take a pragmatic view of how meaning is constructed in natural language. According to Wittgenstein, "the meaning of a word is its use in the language" [27]. In other words, the meaning is simply the sum of all the uses we find of the word — there is no "objective" definition apart from this sum. It follows that meaning is not static, since new examples of use will skew the meaning in their own direction. Also, any attempt at providing a definition for a word (for instance in a dictionary, or our own phrase book for Java [12]) is necessarily an imperfect approximation of the meaning.

Wittgenstein used the term *language game* (Sprachspiel) to designate simple forms of language, "consisting of language and the actions into which it is woven" [27]. Intuitively, a language game should be understood as interplay between natural language expressions and behaviours. Hence, our object of inquiry is really the Java language game, where the language expressions are encoded in method identifiers and the actions are encoded in method bodies.

In discussing the meaning of symbolic language expressions, Frege [9] introduces the terms *sign*, *reference* and *sense*. The sign is the name itself, or a combination of words. The reference is the object to which the sign refers. The sense is our

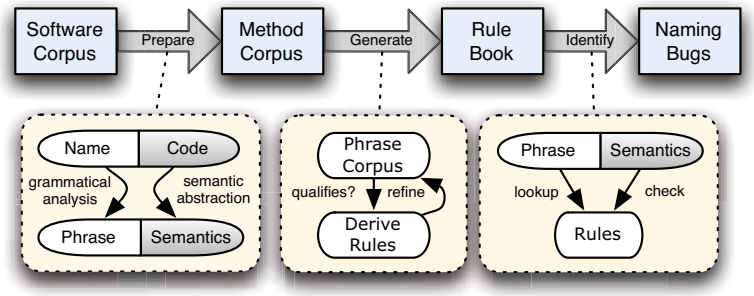


Figure 9.1: Overview of the approach.

collective understanding of the reference. In the context of Java programming, we take the sign to be the method phrase, the reference to be the "true meaning" indicated by that phrase (that Wittgenstein would claim is illusory), and the sense to be the Java community's collective understanding of what the phrase means. Of course, the collective understanding is really unavailable to us: we are left with our own subjective and imperfect understanding of the sign. This is what Frege refers to as the individual's *idea*. Depending on our level of insight, that idea may be in various degrees of harmony or conflict with the actual sense.

Interestingly, when analysing Java methods, we *do* have direct access to a manifestation of the programmer's idea of the method name's sense: the method body. By collecting and analysing a large number of such ideas, we can approximate the sense of the name. This, in turn, allows us to identify naming bugs: ideas that are in conflict with the approximated sense.

9.3 Analysis of Methods

We turn our understanding of how meaning is constructed into a practical approach for approximating the meaning of method names in Java. This approximation is then used to create rules for method implementations. Finally, these rules help us identify naming bugs. Fig. 9.1 provides an overview of the approach. The analysis consists of three major phases: data preparation, mining of implementation rules, and identification of naming bugs.

In the data preparation phase, we transform our corpus of Java applications into an idealised corpus of methods. The transformation entails analysing each Java method in two ways. On the one hand, we perform a natural language analysis on the method name (Sect. 9.3.2). This involves decomposing the name into individual words and performing part-of-speech tagging of those words. The tags allow us to form abstract phrases from the concrete method names. On the other hand, we analyse the signature and Java bytecode of the method implementation, deriving a semantic profile for each implementation (Sect. 9.3.3).

This sets us up to investigate the semantics of methods that share the same abstract phrase. We start with very abstract phrases that we gradually refine into more concrete phrases, more closely matching the actual method names. If a given phrase

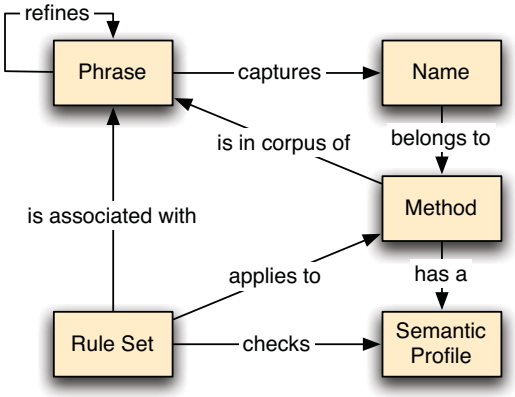


Figure 9.2: Conceptual model of phrase terms.

fulfils certain criteria pertaining to prevalence, we derive a corresponding set of implementation rules (Sect. 9.3.4) that all methods whose names match the phrase must obey. Failure to obey an implementation rule is considered a naming bug (Sects. 9.3.5 and 9.3.6).

9.3.1 Definitions

In the following, please refer to Fig. 9.2 for an overview of the relationships between the introduced terms.

We define a *method* m as a tuple consisting of a unique *fingerprint* u, a *name* n, and a *semantic profile* $[\![m]\!]$. The unique fingerprints prevent set elements from collapsing into one; hence, a set made from arbitrary methods $m_1, \ldots, m_k$ will always have k elements. The name n is a non-empty list of *fragments* f. Each fragment is annotated with a *tag* t.

The semantic profile $[\![m]\!]$ for a method m is defined in terms of *attributes*. We define a set $\mathcal{A}$ of attributes $\{a_1, \ldots, a_k\}$, and let a denote an attribute from $\mathcal{A}$. Given a method m and an attribute a, the expression $check(m, a)$ is a binary value $b \in \{0, 1\}$. Intuitively, *check* determines whether or not m fulfils the predicate defined by a. We then define $[\![m]\!]$ as the list $[check(m, a_1), \ldots, check(m, a_k)]$. It follows that there are at most $2^{|\mathcal{A}|}$ distinct semantic profiles. The *rank* of a semantic profile in a corpus is the proportion of methods that have that semantic profile.

A *phrase* p is a non-empty list of *parts* ρ; its purpose is to abstract over method names. A part ρ may be a fragment f, a tag t, or a special wildcard symbol $*$. The wildcard symbol may only appear as the last part of a phrase. A phrase that consists solely of fragments is *concrete*; all other phrases are *abstract*.

A phrase *captures* a name if each individual part of the phrase captures each fragment of the name, in order from first to last. A fragment part captures a fragment if they are equal. A tag part captures a fragment if it is equal to the fragment's tag. A wildcard part captures any remaining fragments in a name, including zero fragments. A concrete phrase can only capture a single name, whereas an abstract phrase can capture multiple names. For instance, the abstract phrase **is-⟨adjective⟩-*** captures

names like **is-empty**, **is-valid-signature** and so forth.

A *corpus* $\mathcal{C}$ is a set of methods. Implicitly, $\mathcal{C}$ defines a set $\mathcal{N}$, consisting of the names of the methods $m \in \mathcal{C}$. A *name corpus* $\mathcal{C}_n$ is the subset of $\mathcal{C}$ with the name n. Similarly, a *phrase corpus* $\mathcal{C}_p$ is the subset of $\mathcal{C}$ whose names are captured by the phrase p. The *frequency value* $\xi_a(\mathcal{C})$ for an attribute a given a corpus $\mathcal{C}$ is defined as:

$$\xi_a(\mathcal{C}) \stackrel{\text{def}}{=} \frac{\sum_{m \in \mathcal{C}} check(m, a)}{|\mathcal{C}|}$$

The semantics of a corpus $\mathcal{C}$ is defined as the list $[\xi_{a_1}(\mathcal{C}), \ldots, \xi_{a_k}(\mathcal{C})]$. We write $[\![p]\!]_{\mathcal{C}}$ for the semantics of a phrase in corpus $\mathcal{C}$, and define it as the semantics of the corresponding phrase corpus. The subscript will be omitted when there can be no confusion as to which corpus we refer to.

We introduce a subset $\mathcal{A}_o \subset \mathcal{A}$ of *orthogonal attributes*. Two attributes a_1 and a_2 are considered orthogonal if $check(m, a_1)$ does not determine $check(m, a_2)$ or vice versa for any method m. We define the *semantic distance* $d(p_1, p_2)$ between two phrases p_1 and p_2 as the vector distance

$$d(p_1, p_2) \stackrel{\text{def}}{=} \sum_{a \in \mathcal{A}_o} \left(\xi_a(\mathcal{C}_{p_1}) - \xi_a(\mathcal{C}_{p_2})\right)^2$$

A *rule* r is a tuple consisting of an attribute a, a *trigger condition* c and a *severity* s. The trigger condition c is a binary value, indicating whether the rule is triggered when the function *check* evaluates to 0 or to 1. The severity s is defined as $s \in \{forbidden, inappropriate, reconsider\}$.

For example, the rule $r = (a_{reads\_field}, 1, inappropriate)$ indicates that it is considered *inappropriate* for the **reads field** attribute to evaluate to 1. Applied to a method implementation, the rule states that the implementation should not read field values. In practice, rules are relevant for specific phrases. Hence, we associate with each phrase p a set of rules $\mathcal{R}_p$ that apply to the methods $m \in \mathcal{C}_p$.

Finally, we define a boolean function $bug(r, m) \stackrel{\text{def}}{=} check(m, a) = c$ that evaluates to true when the rule $r = (a, c, s)$ is triggered by method m.

9.3.2 Analysing Method Names

Far from being arbitrary labels, method names act as hosts for meaningful phrases. This is the premise we rely on when we state that it is possible to define name-specific rules for the implementation of methods. According to Liblit [15], "[method] names exhibit regularities derived from the grammars of natural languages, allowing them to combine together to form larger pseudo-grammatical phrases that convey additional meaning about the code". To reconstruct these phrases, we decompose the method names into individual fragments, and apply a natural language processing technique called part-of-speech tagging [17] to identify their grammatical structure.

9.3.2.1 Decomposition.

By convention, Java programmers use "camel case" when forming method names that consist of multiple fragments ("words"). A camel case method name uses capitalised fragments to compensate for the lack of whitespace in identifiers. For instance, instead

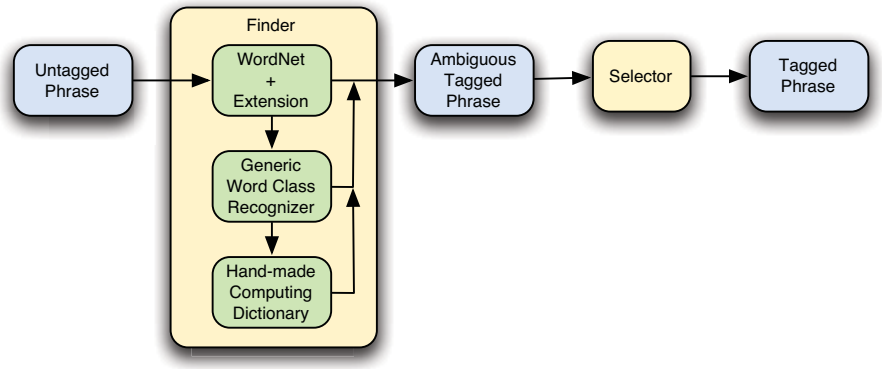


Figure 9.3: Part-of-speech tagging for method phrases.

of writing `create new instance` (which would be illegal), Java programmers write `createNewInstance`. To recover the individual fragments, we reverse the process, using capital characters as an indicator to split the name, with special treatment of uppercase acronyms. For instance, we decompose `parseXMLNode` into `parse XML node` as one would expect. Some programmers use underscore as delimiter instead of case-switching; however, we have previously noted that this is quite rare [12]. For simplicity, we therefore choose to omit such methods from the analysis.

9.3.2.2 Part-of-speech Tagging.

Informally, part-of-speech tagging refers to the process of tagging each word in a natural language expression with information about its the grammatical role in the expression. In our scenario, this translates to tagging each fragment in the decomposed method name. We consider a decomposed method name to be an untagged method phrase.

An overview of the tagging process is shown in Fig. 9.3. First, we use the tags **verb**, **noun**, **adjective**, **adverb**, **pronoun**, **preposition**, **conjunction**, **article**, **number**, **type** and **unknown** to tag each fragment in the phrase. In other words, apart from the special tags **number**, **type** and **unknown**, we use the basic word classes. The **number** tag is used for numeric fragments like **1**. The **type** tag is used when we identify a fragment as the name of a type in scope of the method. Fragments that we fail to tag default to the **unknown** tag.

We make three attempts at finding suitable tags for a fragment. First, we use WordNet [8], a large lexical database of English, to find verbs, nouns, adjectives and adverbs. We augment the results given by WordNet with lists of pronouns, prepositions, conjunctions and articles. If we fail to find any tags, we use a mechanism for identifying invented words. Programmers sometimes derive nouns and adjectives from verbs (for instance, **handler** from **handle** and **foldable** from **fold**), or verbs from nouns (for instance, **tokenize** from **token**). If we can discover such derivations, we tag the fragment accordingly. Finally, we resort to a manual list of tags for commonly used programming terms.

Since a fragment may receive multiple tags (for instance, WordNet considers **object** to be both a noun and a verb), the initial tagging leads to an ambiguously tagged

Figure 9.4: The refinements leading to **is-empty**.

phrase. We then perform a selection of tags that takes into account both the fragment's position in the phrase, and the tags of surrounding fragments. This yields an unambiguously tagged phrase. We have previously estimated the accuracy of the part-of-speech tagger to be approximately 97% [12].

9.3.2.3 Method Phrases and Refinement.

The decomposed, tagged method names are concrete method phrases. The tags allow us to form abstract phrases as well; phrases where concrete fragments have been replaced by tags. Phrases are written like this: **get-⟨noun⟩-***, where the individual parts are separated by hyphens, fragments are written straightforwardly: **get**, tags are written in angle brackets: ⟨**noun**⟩, and the * symbol indicates that the phrase can be further refined.

Refinement involves reducing the corresponding phrase corpus to a subset. In general, there are three kinds of refinement:

1. Introduce tag: **p-*** ⇒ **p-⟨t⟩-***.
   For instance, the phrase **is-*** may be refined to **is-⟨adjective⟩-***. The former phrase would capture a name like `isObject`, the latter would not.

2. Remove wildcard: **p-*** ⇒ **p**.
   For instance, the phrase **is-⟨adjective⟩-*** may be refined to **is-⟨adjective⟩**. The former phrase would capture a name like `isValidSignature`, the latter would not.

3. Replace tag with fragment: **p-⟨t⟩-*** ⇒ **p-f-***.
   For instance, the phrase **is-⟨adjective⟩-*** may be refined to **is-empty-***. The former phrase would capture a name like `isValid`, the latter would not.

Fig. 9.4 shows the refinement steps leading from the completely abstract phrase *, to the concrete phrase **is-empty**. When we reach a concrete phrase, we attempt a final step of further refinement to annotate the concrete phrase with information about the types of return value and parameters. Hence we can form signature-like phrases like **boolean is-empty()**. This step is not included in the figure, nor in the list above.

9.3.3 Analysing Method Semantics

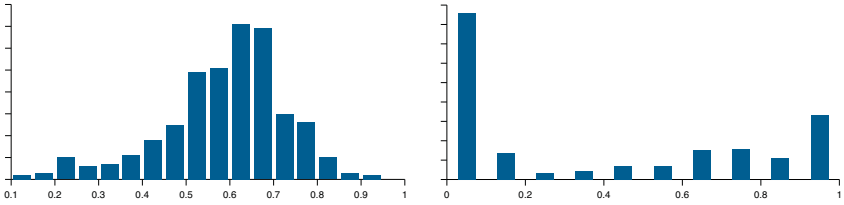
In any data mining task, the outcome of the analysis depends on the domain knowledge of the analyst [26]. Hence, we must rely on our knowledge of Java programming when modelling the semantics of methods. In particular, we consider some aspects of the implementation to be important clues as to the behaviour of methods, whereas others are considered insignificant.

Table 9.1: Attributes. Orthogonal attributes marked with an asterisk.

Signature	
Returns void*	Returns reference
Returns int	Returns boolean
Returns string	No parameters*
Return type in name	Parameter type in name
Data Flow	
Reads field*	Writes field*
Writes parameter value to field	Returns field value
Returns created object	Runtime type check*
Object Creation	
Creates regular objects*	Creates string objects
Creates custom objects	Creates own type objects
Control Flow	
Contains loop*	Contains branch
Multiple return points*	
Exception Handling	
Throws exceptions*	Catches exceptions*
Exposes checked exceptions	
Method Call	
Recursive call*	Same name call*
Same verb call*	Method call on field value
Method call on parameter value	Parameter value passed to method call on field value

A method m has some basic behaviours pertaining to data flow and control flow that we would like to capture: 1) read or write fields, 2) create new objects, 3) return a value to the caller, 4) call methods, 5) branch and/or repeat iteration, and 6) catch and/or throw exceptions. We concretise the basic behaviours by means of a list of machine-traceable *attributes*, formally defined as predicates on Java bytecode. In addition to the attributes stemming from the basic behaviours, called *instruction attributes*, we define a list of *signature attributes*. Table 9.1 lists all the attributes, coarsely sorted in groups. Note that some attributes, such as **returns created object** really belong to more than one group. Attributes marked with an asterisk belong to the subset of orthogonal attributes.

Most of the attributes should be fairly self-explanatory; however, the attributes pertaining to object creation warrant further explanation. A regular object is an object that does not inherit from the type `java.lang.Throwable`, a string object is an instance of the type `java.lang.String`, and a custom object is one that does not belong to either of the namespaces `java.*` and `javax.*`. Finally, the attribute **creates own type objects** indicates that the method creates an instance of the class on which the method is defined.



(a) **Reads field** (b) **Returns void**

Figure 9.5: Distribution of frequency values for two attributes.

9.3.4 Deriving Phrase-Specific Implementation Rules

We derive a set of implementation rules for method phrases that are *prevalent* in a large corpus of Java applications. A phrase is considered prevalent if it fulfils a simple heuristic: it must occur in at least half of the applications in the corpus, and it must cover at least 100 method instances. While somewhat arbitrary, this heuristic guards against idiosyncratic naming in any single application, and ensures a fairly broad basis for the semantics of the phrase. Each prevalent phrase is included in a conceptual "rule book" derived from the corpus, along with a corresponding set of rules. Intuitively, all methods captured by a certain phrase must obey its implementation rules.

We define the implementation rules on the level of individual attributes. To do so, we consider the frequency values of the attributes for different phrase corpora. The intuition is that for a given phrase corpus, the frequency value for an attribute indicates the probability for the attribute's predicate to be fulfilled for methods in that corpus. For each attribute $a \in \mathcal{A}$, we find that the the frequency value $\xi_a(\mathcal{C}_n)$ is distributed within the boundaries $0 \leq \xi_a(\mathcal{C}_n) \leq 1$. We assume that method names therefore can be used to predict whether or not an attribute will evaluate to 1: different names lead to different frequency values. Fig. 9.5 shows example distributions for the attributes **reads field** and **returns void** for some corpus. We see that the two distributions are quite different. Both attributes distinguish between names, but **returns void** is clearly the most polarising of the two for the corpus in question.

A frequency value close to 0 indicates that it is rare for methods in the corresponding corpus to fulfil the predicate defined by the attribute; a value close to 1 indicates the opposite. We exploit this to define rules. Any method that deviates from the norm set by the phrase corpus to which it belongs is suspect. If the norm is polarised (close to 0 or 1), we induce a rule stating that the attribute should indeed evaluate to only the most common value. Breaking a rule constitutes a naming bug. Note that there are two kinds of naming bugs, that we call *inclusion bugs* and *omission bugs*. The former refers to methods that fulfil the predicate of an attribute it should not, the latter to methods that fail to fulfil a predicate it should. We expect inclusion bugs to be more common (and arguably more severe) than omission bugs. For instance, it might be reasonable to refrain from doing anything at all (an empty method) regardless of name, whereas throwing an exception from a seemingly innocent `hasChildren` method is more dubious.

Specifically, we induce rules by defining percentiles on the distribution of frequency

Table 9.2: Percentile groups for frequency values.

Percentiles (%)	*Severity*
0.0	Forbidden (if included)
$0.0 - 2.5$	Inappropriate (if included)
$2.5 - 5.0$	Reconsider (if included)
$5.0 - 95.0$	No violation
$95.0 - 97.5$	Reconsider (if omitted)
$97.5 - 100.0$	Inappropriate (if omitted)
100.0	Forbidden (if omitted)

values for each attribute $a \in \mathcal{A}$. The percentiles are 0.0%, 2.5%, 5.0%, 95.0%, 97.5% and 100.0%, and are associated to a degree of severity when the corresponding rules are violated (see Table 9.3.4). The intuition is that the percentiles classify the frequency values of different phrases relative to each other. Assume, for instance, that we have a corpus $\mathcal{C}$ and a phrase p with a corresponding corpus $\mathcal{C}_p \subset \mathcal{C}$ of methods yielding a frequency value $\xi_a(\mathcal{C}_p)$ for a certain attribute $a \in \mathcal{A}$. Now assume that the frequency value belongs to the lower 2.5% when compared to that of other phrases in $\mathcal{C}$. Then we deem it *inappropriate* for a method $m \in \mathcal{C}_p$ to fulfil the predicate defined by a.

9.3.5 Finding Naming Bugs

Once a set of rules has been obtained for each prevalent phrase in the corpus, finding naming bugs is trivial. For each of the methods we want to check, we attempt to find the rule set for the most concrete capturing phrase (see Fig. 9.2). In a few cases, the capturing phrase may be fully concrete, so that it perfectly matches the method name. This is likely to be the case for certain ubiquitous method names and signatures such as `String toString()` and `int size()`, for instance. In most other cases, we expect the phrase to be more abstract. For instance, for the method name `Element findElement()`, the most concrete capturing phrase might be something like **ref find-**⟨**type**⟩. Failure to find any capturing phrase at all could be considered a special kind of naming bug; that the name itself is rather odd.

When we have found the most concrete capturing phrase p, we obtain the corresponding rule set $\mathcal{R}_p$ that applies to the method. For each rule in the rule set, we pass the rule and the method to the function *bug*. Whenever *bug* returns true, we have a rule violation, and hence a naming bug. Note that a single method might violate several implementation rules, yielding multiple naming bugs.

9.3.6 Fixing Naming Bugs

Naming bugs manifest themselves as violations of phrase-specific implementation rules. A rule violation indicates a conflict between the name and the implementation of a method. There are two ways to resolve the conflict: either we assume that the name is correct and the implementation is broken, or vice versa. The former must be fixed by removing offending or adding missing behaviour. While it is certainly possible to attempt to automate this procedure, it is likely to yield unsatisfactory or even wrong

results. The programmer should therefore attend to this manually, based on warnings from the analysis.

We are more likely to succeed, at least partially, in automating the latter. We propose the following approach to find a suitable replacement name for an implementation that is assumed to be correct. The implementation is represented by a certain semantic profile. Every prevalent phrase that has been used for that profile is considered a *relevant* phrase for replacement. Some of the relevant phrases may be unsuitable, however, because they have rules that are in conflict with the semantic profile. We therefore filter the relevant phrases for rule violations against the semantic profile. The resulting list of phrases are *candidates* for replacement. Note that, in some cases, the list may be empty. If so, we deem the semantic profile to be *unnameable*.

Finding the best candidate for replacement is a matter of sorting the candidate list according to some criterion. We consider three relevant factors: 1) the rank of the semantic profile in the candidate phrase corpus, 2) the semantic distance from the inappropriate phrase to the candidate phrase, and 3) the number of syntactic changes we must apply to the inappropriate phrase to reach the candidate phrase. We assume that the optimal sorting function would take all three factors — and possibly others — into consideration. As a first approximation to solving the problem, however, we suggest simply sorting the list according to profile rank and semantic distances separately, and letting the programmer choose the most appropriate of the two.

9.4 The Corpus

The main requirements for the corpus are as follows:

- It must be representative of real-world Java programming.
- It must cover a variety of applications and domains.
- It must include most well-known and influential applications.
- It must be large enough to be credible as establishing "canonical" use of method names.

Table 9.3 lists the 100 Java applications, frameworks and libraries that constitute our corpus. Building and cleaning a large corpus is time-consuming labour; hence we use the same corpus that we have used in our previous work [12, 11]. The corpus was constructed to cover a wide range of application domains and has been carefully pruned for duplicate code. The only alteration we have made in retrospect is to remove a large number of near-identical code-generated `parse` methods from XBeans and Geronimo. The code clones resulted in visibly skewed results for the **parse-*** phrase, and proves that code generation is a real problem for corpus-based data mining.

Some basic numbers about the pruned corpus are listed in Table 9.4. We omit methods flagged as synthetic (generated by the compiler) as well as methods with "non-standard names". We consider a standard name be at least two characters long, start with a lowercase letter, and not contain any dollar signs or underscores.

Table 9.3: The corpus of Java applications and libraries.

Desktop applications			
ArgoUML 0.24	Azureus 2.5.0	BlueJ 2.1.3	Eclipse 3.2.1
JEdit 4.3	LimeWire 4.12.11	NetBeans 5.5	Poseidon CE 5.0.1
Programmer tools			
Ant 1.7.0	Cactus 1.7.2	Checkstyle 4.3	Cobertura 1.8
CruiseControl 2.6	Emma 2.0.5312	FitNesse	JUnit 4.2
Javassist 3.4	Maven 2.0.4	Velocity 1.4	
Languages and language tools			
ANTLR 2.7.6	ASM 2.2.3	AspectJ 1.5.3	BSF 2.4.0
BeanShell 2.0b	Groovy 1.0	JRuby 0.9.2	JavaCC 4.0
Jython 2.2b1	Kawa 1.9.1	MJC 1.3.2	Polyglot 2.1.0
Rhino 1.6r5			
Middleware, frameworks and toolkits			
AXIS 1.4	Avalon 4.1.5	Google Web Toolkit 1.3.3	JXTA 2.4.1
JacORB 2.3.0	Java 5 EE SDK	Java 6 SDK	Jini 2.1
Mule 1.3.3	OpenJMS 0.7.7a	PicoContainer 1.3	Spring 2.0.2
Sun WTK 2.5	Struts 2.0.1	Tapestry 4.0.2	WSDL4J 1.6.2
Servers and databases			
DB Derby 10.2.2.0	Geronimo 1.1.1	HSQLDB	JBoss 4.0.5
JOnAS 4.8.4	James 2.3.0	Jetty 6.1.1	Tomcat 6.0.7b
XML tools			
Castor 1.1	Dom4J 1.6.1	JDOM 1.0	Piccolo 1.04
Saxon 8.8	XBean 2.0.0	XOM 1.1	XPP 1.1.3.4
XStream 1.2.1	Xalan-J 2.7.0	Xerces-J 2.9.0	
Utilities and libraries			
Batik 1.6	BluePrints UI 1.4	c3p0 0.9.1	CGLib 2.1.03
Ganymed ssh b209	Genericra	HOWL 1.0.2	Hibernate 3.2.1
JGroups 2.2.8	JarJar Links 0.7	Log4J 1.2.14	MOF
MX4J 3.0.2	OGNL 2.6.9	OpenSAML 1.0.1	Shale Remoting
TranQL 1.3	Trove	XML Security 1.3.0	
Jakarta commons utilities			
Codec 1.3	Collections 3.2	DBCP 1.2.1	Digester 1.8
Discovery 0.4	EL 1.0	FileUpload 1.2	HttpClient 3.0.1
IO 1.3.1	Lang 2.3	Modeler 2.0	Net 1.4.1
Pool 1.3	Validator 1.3.1		

Table 9.4: Basic numbers about the corpus.

JAR files	1003
Class files	189941
Candidate methods	1226611
Included methods	1090982

9.5 Results

Here we present results from applying the extracted implementation rules on the corpus, as well as a small set of additional Java applications. In general, the rules can be applied to any Java application or library. For reasons of practicality and scale, however, we focus primarily on bugs in the corpus itself. We explain in detail how the analysis automatically identifies and reports a naming bug, and proceeds to suggest a replacement phrase to use for the method. We then investigate four rather different kinds of naming bugs revealed by the analysis. Finally, we present some overall naming bug statistics, and discuss the validity of the results.

9.5.1 Name Debugging in Practice

We revisit the example method from the introduction, and explain how the analysis helps us debug it.

```
public Field containsField(String name) {
  for (Iterator e = this.field_vec.iterator(); e.hasNext();) {
    Field f = (Field) e.next();
    if (f.getName().equals(name))
      return f;
  }
  return null;
}
```

Recall that we manually identified this as a *naming bug*, since we expect **contains-*** methods to return boolean values. Intuition tells us that *find* would be a more appropriate verb to use.

Finding the Bug. The analysis successfully identifies this as a naming bug, in the following way. First, we analyse the method. The name is decomposed into the fragments "contains" and "Field", which are tagged as **verb** and **type**, respectively. From the implementation, we extract a semantic profile that has the following attributes from Table 9.1 evaluated to 1, denoting presence: **return type in name**, **reads field**, **run-time type-check**, **contains loop**, **has branches**, **multiple returns**, **method call on field**. The rest of the attributes are evaluated to 0, denoting absence. We see that the attributes conspire to form an abstract description of the salient features of the implementation.

The most suitable phrase in our automatically generated rule book corresponding to the concrete phrase **contains-Field** is the abstract phrase **contains-***. The rule set for **contains-*** is listed in Table 9.5, along with the violations for the semantic profile. The mismatch between the name and implementation in this case manifests itself as three naming bugs. A **contains-*** should not return a reference type (much less echo the name of that type in the name of the method); rather, it should return a boolean value.

Fixing the Bug. There are two ways to fix a naming bug; either by changing the implementation, i.e., by returning a boolean value if the `Field` is found (rather than

Table 9.5: Rules for **contains-*** methods.

Attribute	*Condition*	*Severity*	*Violation*
Returns void	1	Forbidden	No
Returns boolean	0	Inappropriate	Yes
Returns string	1	Inappropriate	No
Returns reference	1	Reconsider	Yes
Return type in name	1	Inappropriate	Yes
Parameter type in name	1	Reconsider	No
Writes field	1	Reconsider	No
Returns created object	1	Forbidden	No
Creates own class objects	1	Inappropriate	No

Table 9.6: Candidate replacement phrases.

Phrase	*Distance*	*Rank*	*Sum*
find-⟨type⟩	4	3	7
find-*	2	5	7
ref find-⟨type⟩	7	1	8
find-⟨type⟩-*	5	4	9
find-⟨adjective⟩-*	3	6	9
ref find-⟨type⟩-*	8	2	10
find-⟨noun⟩-*	1	9	10
get-⟨type⟩-*(String...)	6	8	14
ref get-⟨type⟩-*(String...)	9	7	16
ref get-⟨type⟩-*	10	10	20

the `Field` itself), or by changing the name. In Sect. 9.3.6 we describe the approach for automatic suggestion of bug-free method names, to assist in the latter scenario.

Consider the top ten candidate replacement phrases listed in Table 9.6. An immediate reaction is that the candidates are fairly similar, and that *all* of them seem more appropriate than the original. Here we have sorted the list according to the sum of the orders given by the two ordering metrics *semantic distance* and *profile rank*; in cases of equal sum, we have arbitrarily given precedence to the phrase with the highest rank. In this particular example, we see that a rank ordering gives the better result, by choosing **ref find-⟨type⟩** over the more generic **find-⟨noun⟩-***.

9.5.2 Notable Naming Bugs

To illustrate the diversity of naming bugs the phrase-specific implementation rules help us find, we explore a few additional examples of naming bugs found in the corpus. The four methods shown in Fig. 9.6 exhibit rather different naming bugs. Note that since both strategies for choosing replacement phrases yield similar results, we have included only the top candidate according to profile rank in the figure.

The first example, taken from Ant 1.7.0, is representative of a fairly common naming bug: the inappropriately named "boolean setter". While both Java convention and the JavaBean specification[2] indicate that the verb *set* should be used for all methods for writing properties (including boolean ones), programmers sometimes use an inappropriate **is-*** form instead. This mirrors convention in some other languages such as Objective-C, but yields the wrong expectation when programming Java. The problem is, of course, that `isCaching` reads like a question: "is it true that you are caching?". We expect the question to be answered. The analysis indicates three rule violations for the method, and suggests using the phrase **set-⟨adjective⟩-*** instead.

The second example, taken from the class `Value` in JXTA 2.4.1, shows a broken implementation of an `equals` method. According to Sun's documentation, "The equals method implements an equivalence relation on non-null object references"[3]: it should be reflexive, symmetric, transitive and consistent. It turns out that this is notoriously hard to implement correctly. An influential book on Java devotes much attention to the details of fulfilling this contract [3]. The problem with the implementation from JXTA is that it is *not* symmetric, and the symptom is the creation of an instance of the type that defines the method. Assume that we have a `Value` instance v. The last instruction returns true whenever the parameter can be serialised to a String that in turn is used to create a `Value` object that is equal to v. For instance, we can get a `true` return value if we pass in a suitable String object s. However, if we pass v to the `equals` method of s, we will get `false`. Interestingly, we find no appropriate replacement phrase for this method. This is good news, since it makes little sense to rename a broken `equals` method.

The third example, an `iterator` method from the class `Registry` in Java 5 Enterprise Edition, illustrates the problem of overloading and redefining a well-established method name. The heavily implemented `Iterable<T>` interface defines a method signature `Iterator<T> iterator()`. Since the signature does not include any checked exceptions, the expectation naturally becomes that `iterator` methods in general do not expose any checked exceptions — indeed, the compiler will stop implementors of

[2] `http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html`
[3] `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html`

<table>
<tr>
<td>

```
// Ant 1.7.0
public void isCaching(boolean value) {
   this.caching = value;
}
```

</td>
<td>
Semantic Profile:
Returns void, Writes field, Parameter to field

Violated rules for 'is-<adjective>-*'
Returns void: Reconsider if included
Returns boolean: Inappropriate if missing
Parameter to field: Inappropriate if included

Replacement phrase:
'set-<adjective>-*'
</td>
</tr>
<tr>
<td>

```
// JXTA 2.4.1
public boolean equals(Object obj) {
   if (this == obj)
      return true;
   if (obj instanceof Value)
      return equals((Value)obj);
   return equals(new Value(obj.toString()));
}
```

</td>
<td>
Semantic Profile:
Returns boolean, Runtime type-check, Has branches, Multiple returns, Creates regular objects, Creates custom objects, Creates own class objects, Same name call, Method call on parameter

Violated rules for 'boolean equals(Object...)
Creates own class objects: Reconsider if included

Replacement phrase:
--- None ---
</td>
</tr>
<tr>
<td>

```
// Java 5 EE SDK
public Iterator iterator()
throws DomainRegistryException {
   return new RegistryIterator(this, this);
}
```

</td>
<td>
Semantic Profile:
Returns reference, No parameters, Return type in name, Returns created object, Creates regular objects, Creates custom objects, Exposes checked exceptions

Violated rules for 'ref iterator()'
Exposes checked exceptions: Notify if included

Replacement phrase:
'create-<adjective>-<noun>'
</td>
</tr>
<tr>
<td>

```
// DB Derby 10.2.2.0
public OutputStream setBinaryStream(long val)
throws SQLException {
   checkValidity();
   synchronized (this.agent_.connection_) {
      // Logging code removed.
      BlobOutputStream result = new
         BlobOutputStream(this, val);
      // Logging code removed.
      return result;
   }
}
```

</td>
<td>
Semantic Profile:
Returns reference, Reads field, Returns created object, Has branches, Creates regular objects, Exposes checked exceptions, Method call on field

Violated rules for 'set-<adjective>-*'
Returns created object: Inappropriate if included

Replacement phrase:
'open-*'
</td>
</tr>
</table>

Figure 9.6: Four notable naming bugs from the corpus.

`Iterable<T>` if they try. However, `Registry` does not implement `Iterable<T>`, it simply uses a similar signature. But it is a bad idea to do so, since violating expectations is bound to cause confusion. It is particularly troublesome that the implementation exposes a checked exception, since this is something *iterator* methods practically never do. Note that the replacement phrase makes perfect sense since the method acts as a *factory* that creates new objects.

The final example is a bizarrely named method from DB Derby 10.2.2.0: clearly this is no *setter*! The semantic profile of the method is complicated a bit by the synchronisation and logging code, but for all intents and purposes, this is a factory method of sorts. The essential behaviour is that an object is created and returned to the caller. Creating and returning objects is inappropriate behaviour for methods that match the phrase **set-⟨adjective⟩-*'**; hence we get a rule violation. The suggested replacement phrase, **open-***, is not completely unreasonable, and certainly better than the original.

9.5.3 Naming Bug Statistics

We now consider the more general prevalence of naming bugs. Table 9.7 presents naming bug statistics for all the applications in the corpus, as well as a small number of additional applications. The additional applications are placed beneath a horizontal line near the bottom of the table. For each application, we list the number of methods, the percentage of those methods covered by implementation rules, and the percentage of covered methods violating an implementation rule. We see that the naming bug rates are fairly similar for applications in and outside the corpus, suggesting that the rules can meaningfully be applied to any Java application. It is worth noting that the largest applications (for instance, Java, Eclipse and NetBeans) to some extent have the power to dictate what is common usage. At the same time, such applications are developed by many different programmers over a long period of time, making diversity more likely.

It is important to remember that the numbers really indicate how canonical the method implementations are with respect to the names used. Herein lies an element of conformity as well. The downside is that some applications might be punished for being too "opinionated" about naming. For instance, JUnit 4.2 is written by programmers who are known to care about naming, yet the reported naming bug rate, 3.50%, is fairly high. We believe this is due to the tension between maintaining the status quo and trying to improve it.

Where to draw the line between appropriate and inappropriate usage of names is a pragmatic choice, and a trade-off between false positives and false negatives. A narrow range for appropriate usage increases the number of false positives, a broad range increases the number of false negatives. We are not too concerned with false negatives, since our focus is on demonstrating the existence of naming bugs, rather than finding them all. False positives, on the other hand, could pose a threat to the usefulness of our results.

False positives, i.e., that the analysis reports a naming bug that we intuitively disagree with, might occur for the following reasons:

- The corpus may contain noise that leads to rules that are not in harmony with the intuitions of Java programmers.

Table 9.7: Naming bug statistics.

Application	*Methods*	*Covered*	*Buggy*	*Application*	*Methods*	*Covered*	*Buggy*
ANTLR 2.7.6	1641	61.66%	1.18%	ASM 2.2.3	724	45.30%	0.30%
AXIS 1.4	4290	91.35%	1.65%	Ant 1.7.0	7562	89.35%	0.85%
ArgoUML 0.24	13312	81.17%	0.85%	AspectJ 1.5.3	24976	74.41%	1.24%
Avalon 4.1.5	280	82.14%	2.17%	Azureus 2.5.0	14276	78.32%	1.30%
Batik 1.6	9304	85.90%	0.76%	BSF 2.4.0	274	77.37%	0.00%
BeanShell 2.0 Beta	907	74.97%	0.73%	BlueJ 2.1.3	3369	82.13%	1.48%
BluePrints UI 1.4	662	89.57%	0.67%	C3P0 0.9.1	2374	83.06%	1.52%
CGLib 2.1.03	675	80.29%	1.66%	Cactus 1.7.2	3004	87.61%	1.36%
Castor 1.1	5094	91.44%	0.88%	Checkstyle 4.3	1350	76.07%	0.09%
Cobertura 1.8	328	82.92%	1.47%	Commons Codec 1.3	153	79.08%	0.00%
Commons Collections 3.2	2914	77.93%	1.14%	Commons DBCP 1.2.1	823	88.69%	1.09%
Commons Digester 1.8	371	79.24%	0.34%	Commons Discovery 0.4	195	92.30%	0.00%
Commons EL 1.0	277	59.20%	4.87%	Commons FileUpload 1.2	123	91.86%	0.88%
Commons HttpClient 3.0.1	1071	88.98%	1.46%	Commons IO 1.3.1	357	81.23%	5.17%
Commons Lang 2.3	1627	82.72%	1.93%	Commons Modeler 2.0	376	93.35%	1.42%
Commons Net 1.4.1	726	69.69%	1.58%	Commons Pool 1.3	218	71.55%	0.00%
Commons Validator 1.3.1	443	88.03%	1.02%	CruiseControl 2.6	5479	87.18%	0.85%
DB Derby 10.2.2.0	15470	80.08%	2.09%	Dom4J 1.6.1	1645	92.15%	0.39%
Eclipse 3.2.1	110904	81.65%	1.03%	Emma 2.0.5312	1105	82.62%	0.65%
FitNesse	2819	74.49%	2.14%	Ganymed ssh build 209	424	76.65%	1.23%
Genericra	454	86.78%	0.50%	Geronimo 1.1.1	26753	85.28%	0.71%
Google WT 1.3.3	4129	73.40%	1.78%	Groovy 1.0	10237	76.14%	1.01%
HOWL 1.0.2	173	81.50%	1.41%	HSQLDB	3267	86.16%	2.98%
Hibernate 3.2.1	11354	80.47%	2.00%	J5EE SDK	148701	83.56%	1.17%
JBoss 4.0.5	34965	84.69%	0.95%	JDOM 1.0	144	80.55%	0.86%
JEdit 4.3	3330	80.36%	1.30%	JGroups 2.2.8	4165	77.52%	2.04%
JOnAS 4.8.4	30405	81.88%	1.16%	JRuby 0.9.2	7748	76.69%	1.27%
JUnit 4.2	365	62.46%	3.50%	JXTA 2.4.1	5210	86.96%	1.30%
JacORB 2.3.0	8007	71.01%	1.16%	James 2.3.0	2382	79.21%	1.85%
Jar Jar Links 0.7	442	53.84%	0.42%	Java 6 SDK	80292	81.03%	1.16%
JavaCC 4.0	370	77.02%	2.80%	Javassist 3.4	1842	84.03%	1.42%
Jetty 6.1.1	15177	73.54%	1.06%	Jini 2.1	8835	80.00%	1.38%
Jython 2.2b1	3612	72.09%	1.65%	Kawa 1.9.1	6309	65.36%	2.01%
Livewire 4.12.11	12212	81.96%	1.15%	Log4J 1.2.14	1138	83.39%	0.63%
MJC 1.3.2	4957	73.77%	1.72%	MOF	28	100.00%	0.00%
MX4J 3.0.2	1671	85.33%	1.26%	Maven 2.0.4	3686	84.69%	0.86%
Mule 1.3.3	4725	86.79%	1.09%	NetBeans 5.5	113355	87.60%	0.85%
OGNL 2.6.9	502	88.24%	0.45%	OpenJMS 0.7.7 Alpha	3624	85.89%	0.70%
OpenSAML 1.0.1	306	92.48%	1.76%	Piccolo 1.04	559	77.10%	0.46%
PicoContainer 1.3	435	67.81%	1.35%	Polyglot 2.1.0	3521	67.33%	1.64%
Poseidon CE 5.0.1	25739	77.73%	1.19%	Rhino 1.6r5	2238	77.56%	1.67%
Saxon 8.8	6596	73.12%	1.22%	Shale Remoting 1.0.3	96	72.91%	0.00%
Spring 2.0.2	8349	88.05%	1.52%	Struts 2.0.1	6106	88.97%	1.06%
Sun Wireless Toolkit 2.5	20538	80.37%	1.59%	Tapestry 4.0.2	3481	78.71%	0.87%
Tomcat 6.0.7 Beta	5726	88.31%	0.90%	TranQL 1.3	1639	77.85%	1.17%
Trove 1.1b4	3164	82.01%	0.23%	Velocity 1.4	3635	81.62%	0.67%
WSDL4J 1.6.2	651	94.16%	0.00%	XBean 2.0.0	7000	81.10%	1.33%
XML Security 1.3.0	819	86.56%	1.55%	XOM 1.1	1399	77.05%	1.85%
XPP 1.1.3.4	426	84.50%	1.38%	XStream 1.2.1	916	77.83%	0.84%
Xalan-J 2.7.0	14643	81.38%	1.21%	Xerces-J 2.9.0	590	89.15%	0.19%
FindBugs 1.3.6	7688	72.78%	1.42%	iText 2.1.4	4643	85.18%	1.54%
Lucene 2.4.0	2965	74.16%	1.50%	Mockito 1.6	1408	68.32%	1.35%
ProGuard 4.3	4148	45.34%	2.65%	Stripes 1.5	1600	89.31%	2.09%

- Some legitimate sub-use of a commonly used phrase may be deemed inappropriate because the sub-use is drowned by the majority. (Arguably a new phrase should be invented to cover the sub-use.)
- The percentiles used to classify attribute fraction rank (Sect. 9.3.4) can be skewed.

Whether or not something classifies as a naming bug is subjective. What is *not* subjective, is the fact that all reported issues will be rare, and therefore worthy of reconsideration. To discern false positives from genuine naming bugs, we must rely on our on best judgement. To get an idea of the severity of the problem, we manually investigated 50 reported naming bugs chosen at random. We found that 30% of the reported naming bugs in the sample were false positives, suggesting that the approach holds promise (even though, due to the limited size of the sample, the true false positive rate might be significantly higher or lower). The false positives were primarily *getters* that were slightly complex, but not inappropriately so in our eyes, and methods containing logging code.

9.5.4 Threats to Validity

There are three major threats to the validity of our results:

- Does the pragmatic view of how meaning is constructed apply to Java programming?
- Is the corpus representative of real-world Java programming?
- Is the attribute model a suitable approximation of the actual semantics of a method?

Our basic assumption is that canonical usage of a method name is also meaningful and appropriate usage; this relates to the pragmatic view that meaning stems from actual use. We establish the meaning of phrases using a crude democratic process of voting. This approach is not without problems. First, it is possible for individual idiosyncratic applications to skew the election. In particular, code generation can lead to problems, since it enables the proliferation of near-identical clones. While we can spot gross examples of this (see Sect. 9.4), code generation on a smaller scale is hard to detect, and can affect the results for individual phrases. This in turn can corrupt our notion of canonical usage, leading to corrupt rules and incorrect reports of naming bugs. Second, there might be individual applications that use a language that is both richer, more consistent and precise than the one used by the majority. However, the relative uniformity in the distribution of naming bugs seems to indicate that neither of these problems are too severe. Despite these problems, therefore, we believe that the pragmatic view of meaning applies well to Java programming. It is certainly more reasonable to use the aggregated ideas of many as an approximation of meaning than to make an arbitrary choice of a single application's idea.

When aggregating ideas, however, we must assume that the ideas we aggregate are representative. The business journalist Surowiecki argues that diversity of opinion, independence, decentralisation and an aggregation mechanism are the prime prerequisites to make good group decisions [25]. The corpus we use was carefully constructed to contain a wide variety of applications and libraries of various sizes and from many

domains. We therefore believe it to fulfil Surowiecki's prerequisites and be reasonably representative of real-world Java programming.

Finally, we consider the suitability of the model for method semantics, which is a coarse approximation based on our knowledge of Java programming. Using attributes to characterise methods has several benefits, in particular that it reduces the practically endless number of possible implementations to a finite set of semantic profiles. Furthermore, the validation of a useful model must come in the form of useful results. As we have seen, the model has helped us identify real naming bugs with what appears to be a relatively low rate of false positives. We therefore believe that the model is adequate for the task at hand.

9.6 Related Work

Micro patterns, introduced by Gil and Maman [10], are a central source of inspiration for our work. Micro patterns are machine-traceable patterns on the level of Java classes. A pattern is machine-traceable if it can be expressed as a simple formal condition on some aspect of a software module. The presented micro patterns are hand-crafted by the authors to capture their knowledge of Java programming.

In our work, we use hand-crafted machine-traceable attributes to model the semantics of methods rather than classes. The attributes are similar to *fingerprints*, a notion used by the Sourcerer code search engine [1]. According to the Sourcerer website[4], the engine supports three kinds of fingerprint-based search, utilising control flow, Java type and micro pattern information respectively. Ma et al. [16] provide a different take on the task of searching for a suitable software artefact. They share our assumption that programmers usually choose appropriate names for their implementations, and therefore use identifier information to index the Java API for efficient queries.

Overall, there seems to be a growing interest in harnessing the knowledge embedded in identifiers. Pollock et al. [20] introduce the term *Natural Language Program Analysis* (NLPA) to signify program analysis that exploits natural language clues. The analysis has been used to develop tools for program navigation and aspect mining [23, 22]. The tools exploit the relationship between natural language expressions in source code (identifiers and comments) and information about the structure of the code.

Singer and Kirkham [24] investigate which type names are used for instances of micro patterns in a large corpus of Java applications. More precisely, the *suffixes* of the actual type names are used (the last *fragment* of the name in our terminology). The empirical results indicate that type name suffixes are indeed correlated to the presence of micro patterns in the code.

Caprile and Tonella [4] analyse the structure of function identifiers in C programs. The identifiers are decomposed into fragments that are then classified into seven lexical categories. The structure of the function identifiers are further described by a hand-crafted grammar.

Lawrie et al. [13] study the quality of identifiers in a large corpus of applications written in several languages. An identifier is assumed to be of high quality if it can be composed of words from a dictionary and well-known abbreviations. This is a better quality indicator than mere uniformity of lexical syntax, but does not address the issue of *appropriateness*. Deißenböck and Pizka [6] develop a formal model for identifier

[4] `http://sourcerer.ics.uci.edu/`

quality, based on *consistency* and *conciseness*. Unfortunately, this model requires an expert to perform manual mapping between identifiers and domain concepts.

Reiss [21] proposes an automatic approach for finding unusual code. The assumption is that unusual code is potentially problematic code. The approach works by mining common syntactic code patterns from a corpus of applications. Unusual code is code that is not covered by such patterns. Hence we see that there are similarities to our work, both in the assumption and the approach. A main difference is that we define unusual code in the context of a given method phrase.

9.7 Conclusion

Natural language expressions get their meaning from how and when they are used in practice. Deviation from normal use of words and phrases leads to misunderstanding and confusion. In the context of software this is particularly bad, since precise understanding of the code is paramount for successful development and maintenance. We have therefore coined the term *naming bug* to describe unusual aspects of implementations for a given method name. We have presented a practical approach to *debugging method names*, by offering assistance both in finding and fixing naming bugs. To find naming bugs, we use name-specific implementation rules mined from a large corpus of Java applications. Naming bugs can be fixed either by changing the implementation or by using a different method name; for the latter task, we have also shown an approach to provide automatic assistance. To demonstrate that method name debugging is useful, we have applied the rules to uncover naming bugs both in the corpus itself and in other applications.

In this and previous work, we have exploited the fact that there is a shared vocabulary of terms and phrases, *Java Programmer English* [12], that programmers use in method names. In the future, we would like to investigate the adequacy of that vocabulary. In particular, there might be terms or phrases that are superfluous, while others are missing, at least from the common vocabulary of Java programmers. We know that there exists verbs (for instance *create* and *new*) that seem to be used almost interchangeably in method names. Our results reveal hints of this, by finding a shorter semantic distance between phrases that use such verbs. By analysing the corresponding method implementations, we could find out whether there are subtle differences in meaning that warrant the existence of both verbs in Java Programmer English. If not, it would be beneficial for Java programmers to choose one and eliminate or redefine the other. There are also verbs (and phrases) that are imprecise, in that they are used to represent many different kinds of implementations. For instance, the ubiquitous *getter* is much less homogenous than one might expect [11], indicating that it has a wide variety of implementations. It would be interesting to see if the verbs are simply used as easy resorts when labelling more or less random chunks of code, or if there are legitimate, identifiable sub-uses that would warrant the invention of new verbs. Or it might be that a minority of the Java community already has invented the proper verbs, and that they should be more widely adopted to establish a richer, more expressive language for all Java programmers to use.

Acknowledgements. We thank Jørn Inge Vestgården, Wolfgang Leister and Truls Fretland for useful comments and discussions, and the anonymous reviewers for their

thoughtful remarks.

Bibliography

[1] S. K. Bajracharya, T. C. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. V. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In P. L. Tarr and W. R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 681–682. ACM, 2006.

[2] K. Beck. *Implementation Patterns*. Addison-Wesley Professional, 2007.

[3] J. Bloch. *Effective Java*. Prentice Hall, 2008.

[4] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE 1999), 6-8 October 1999, Atlanta, Georgia, USA*, pages 112–122. IEEE Computer Society, 1999.

[5] E. Collar and R. Valerdi. Role of software readability on software development cost. In *Proceedings of the 21st Forum on COCOMO and Software Cost Modeling, October 2006, Herndon, VA.*, 2006.

[6] F. Deißenböck and M. Pizka. Concise and consistent naming. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005)*, pages 97–106. IEEE Computer Society, 2005.

[7] M. A. Eierman and M. T. Dishaw. The process of software maintenance: A comparison of object-oriented and third-generation development languages. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(1):33–47, 2007.

[8] C. Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[9] G. Frege. On sense and reference. In P. Geach and M. Black, editors, *Translations from the Philosophical Writings of Gottlob Frege*, pages 56–78. Blackwell, 1952.

[10] J. Gil and I. Maman. Micro patterns in Java code. In R. E. Johnson and R. P. Gabriel, editors, *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), October 16-20, 2005, San Diego, CA, USA*, pages 97–116. ACM, 2005.

[11] E. W. Høst and B. M. Østvold. The programmer's lexicon, volume I: The verbs. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 193–202, Paris, France, 2007. IEEE Computer Society.

[12] E. W. Høst and B. M. Østvold. The Java programmer's phrase book. In *Proceedings of the 1st International Conference on Software Language Engineering (SLE 2008)*. Springer, 2008.

[13] D. Lawrie, H. Feild, and D. Binkley. Quantifying identifier quality: An analysis of trends. *Journal of Empirical Software Engineering*, 12(4):359–388, August 2007.

[14] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? A study of identifiers. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pages 3–12. IEEE Computer Society, 2006.

[15] B. Liblit, A. Begel, and E. Sweeser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, Sussex, United Kingdom, September 2006. Psychology of Programming Interest Group.

[16] H. Ma, R. Amor, and E. D. Tempero. Indexing the Java API using source code. In *Proceedings of the 19th Australian Software Engineering Conference (ASWEC 2008), March 25-28, 2008, Perth, Australia*, pages 451–460. IEEE Computer Society, 2008.

[17] C. D. Manning and H. Schuetze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[18] R. C. Martin. *Clean Code*. Prentice Hall, 2008.

[19] S. McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2nd edition, 2004.

[20] L. L. Pollock, K. Vijay-Shanker, D. Shepherd, E. Hill, Z. P. Fry, and K. Maloor. Introducing natural language program analysis. In M. Das and D. Grossman, editors, *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007), San Diego, California, USA, June 13-14, 2007*, pages 15–16. ACM, 2007.

[21] S. P. Reiss. Finding unusual code. In *Proceedings of the 23rd IEEE International Conference of Software Maintenance (ICSM 2007)*, pages 34–43. IEEE Computer Society, 2007.

[22] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development (AOSD 2007)*, pages 212–224, New York, NY, USA, 2007. ACM.

[23] D. Shepherd, L. L. Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In R. E. Filman, editor, *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006), Bonn, Germany, March 20-24, 2006*, pages 3–14. ACM, 2006.

[24] J. Singer and C. Kirkham. Exploiting the correspondence between micro patterns and class names. In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008)*, pages 67–76, Beijing, China, 2008. IEEE Computer Society.

[25] J. Surowiecki. *The Wisdom of Crowds*. Anchor, 2005.

[26] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2nd edition, 2005.

[27] L. Wittgenstein. *Philosophical Investigations*. Prentice Hall, 1973.

Chapter 10

Paper 4: Canonical Method Names For Java

Programmers rely on the conventional meanings of method names when writing programs. However, these conventional meanings are implicit and vague, leading to various forms of ambiguity. This is problematic since it hurts the readability and maintainability of programs. Java programmers would benefit greatly from a more well-defined vocabulary. Identifying synonyms in the vocabulary of verbs used in method names is a step towards this goal. By rooting the meaning of verbs in the semantics of a large number of methods taken from real-world Java applications, we find that such synonyms can readily be identified. To support our claims, we demonstrate automatic identification of synonym candidates. This could be used as a starting point for a manual canonicalisation process, where redundant verbs are eliminated from the vocabulary.

10.1 Introduction

Abelson and Sussman [1] contend that "programs must be written for people to read, and only incidentally for machines to execute". This is sound advice backed by the hard reality of economics: maintainability drives the cost of software systems [7], and readability drives the cost of maintenance [3, 23]. Studies indicate some factors that influence readability, such as the presence or absence of abbreviations in identifiers [14]. Voices in the industry would have programmers using "good names" [16, 2], typically meaning very explicit names. A different approach with the same goal is *spartan programming*[1]. "Geared at achieving the programming equivalent of laconic speech", spartan programming suggests conventions and practical techniques to reduce the complexity of program texts.

We contend that both approaches attempt to fight *ambiguity*. The natural language dimension of program texts, that is, the expressions encoded in the identifiers of the program, is inherently ambiguous. There are no enforced rules regarding the meaning of the identifiers, and hence we get ambiguity in the form of synonyms (several words are used for a single meaning) and polysemes (a single word has multiple meanings). This ambiguity could be reduced if we managed to establish a more well-defined vocabulary

[1] `http://ssdl-wiki.cs.technion.ac.il/wiki/index.php/Spartan_programming`

for programmers to use.

We restrict our attention to the first lexical token found in method names. For simplicity, we refer to all these tokens as "verbs", though they need not actually be verbs in English: `to` and `size` are two examples of this. We focus on verbs because they form a central, stable part of the vocabulary of programmers; whereas nouns tend to vary greatly by the domain of the program, the core set of verbs stays more or less intact.

We have shown before [10, 11, 12] that the meaning of verbs in method names can be modelled by abstracting over the bytecode of the method implementations. This allows us to 1) identify what is typical of implementations that share the same verb, and 2) compare the set of implementations for different verbs. In this paper, we aim at improving the core vocabulary of verbs for Java programmers by identifying potential synonyms that could be unified.

The contributions of this paper are:

- The introduction of *nominal entropy* as a way to measure how "nameable" a method is (Section 10.3.1).
- A technique to identify methods with "unnameable semantics" based on nominal entropy (Section 10.4.3).
- A technique to mechanically identify likely instances of code generation in a corpus of methods (Section 10.4.1).
- A formula to guide the identification of synonymous verbs in method names (Section 10.3.3).
- A mechanically generated graph showing synonym candidates for the most commonly used verbs in Java (Section 10.5.1).
- A mechanically generated list of suggestions for canonicalisation of verbs through unsupervised synonym elimination (Section 10.5.2).

10.2 Problem description

To help the readability and learnability of the scripting language PowerShell, Microsoft has defined a standardised set of verbs to use. The verbs and their definitions can be found online[2], and PowerShell programmers are strongly encouraged to follow the conventions. The benefits to readability and learnability are obvious.

By contrast, the set of verbs used in method names in Java has emerged organically, as a mixture of verbs inherited from similar preceding languages, emulation of verbs used in the Java API, and so forth. A similar organic process occurs in natural languages. Steels argues that language "can be viewed as a complex adaptive system that adapts to exploit the available physiological and cognitive resources of its community of users in order to handle their communicative challenges" [22].

We have seen before that Java programmers have a fairly homogenous, shared understanding of many of the most prevalent verbs used in Java programs [10]. Yet the organic evolution of conventional verb meaning has some obvious limitations:

[2] `http://msdn.microsoft.com/en-us/library/ms714428%28VS.85%29.aspx`

- **Redundancy**. There are concepts that evolution has not selected a single verb to represent. This leads to superfluous synonymous verbs for the programmer to learn. Even worse, some programmers may use what are conventional synonyms in subtly different meanings.

- **Coarseness.** It is hard to organically grow verbs with precise meanings. To make sure that a verb is understood, it may be tempting to default to a very general and coarse verb. This results in "bagging" of different meanings into a small set of polysemous verbs.

- **Vagueness.** Evolution in Java has produced some common verbs that are almost devoid of meaning (such as `process` or `handle`), yet are lent a sense of legitimacy simply because they are common and shared among programmers.

Redundancy is the problem of *synonyms*, and can be addressed by identifying verbs with near-identical uses, and choosing a single, canonical verb among them. Coarseness is the problem of *polysemes*, and could be addressed by using data mining to identify common polysemous uses of a verb, and coming up with more precise names for these uses. Vagueness is hard to combat directly, as it is a result of the combination of a lack of a well-defined vocabulary with the programmer's lacking ability or will to create a clear, unambiguous and nameable abstraction. In this paper, we primarily address the problem of redundancy.

10.3 Analysis of methods

The meaning of verbs in method names stems from the implementations they represent. That is, the meaning of a verb is simply the collection of observed uses of that verb (Section 10.3.1). Further, we hold that the verbs become more meaningful when they are consistently used to represent similar implementations. To make it easier to compare method implementations, we employ a coarse-grained semantic model for methods, based on predicates defined on Java bytecode (Section 10.3.2). We apply entropy considerations to measure both how consistently methods with the same verb are implemented, and how consistently the same implementation is named. We refer to this as semantic and nominal entropy, respectively. These two metrics are combined in a formula that we use to identify synonymous verbs (Section 10.3.3). Figure 10.1 presents an overview of the approach.

10.3.1 Definitions

We define a *method* m as a tuple consisting of three *components*: a unique *fingerprint* u, a *name* n, and a *semantics* s. Intuitively m is an idealised method, a model of a real method in Java bytecode. The unique fingerprints are a technicality that prevents set elements from collapsing into one; hence, a set made from arbitrary methods $\{m_1, \ldots, m_k\}$ will always have k elements.[3] Often, we elide u from method tuples, writing just $m = (n, s)$.

[3]The fingerprints models the mechanisms that the run-time system has for identifying distinct callable methods.

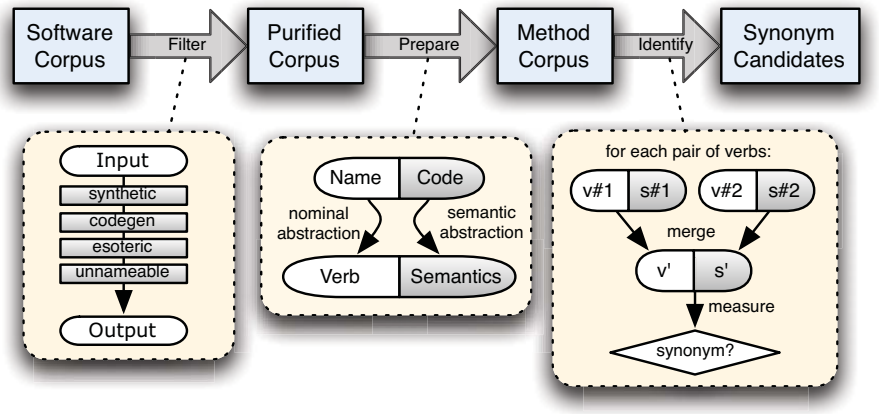


Figure 10.1: Overview of the approach.

We need two kinds of languages to reason about methods and their semantics. First, a concrete language where the semantics of a method is simply the string of Java bytecodes in m's implementation. Thus, bytecode is the canonical *concrete language*, denoted as $\mathcal{L}_{\text{Java}}$. For convenience, we define a labelling function f_{MD5} that maps from the bytecode to the MD5 digest of the opcodes in the bytecode. This allows us to easily apply uniform labels to the various implementations.

Second, we need an abstract language consisting of bit-vectors $[b_1, \ldots, b_k]$ where each b_i represents the result of evaluating a logical predicate q_i on a method's implementation. In the context of a concrete method m and its implementation, we refer to the vector $[b_1, \ldots, b_k]$ as *profile* of m. Different choices of predicates $q_1, \ldots, q_k$, leads to different *abstract languages*. Note that with the concrete language there is no limit on the size of a method's semantics; hence there is in principle an unlimited number of semantic objects. With an abstract language there is a fixed number of semantic objects, since s is a k-bit vector for some fixed number k, regardless of the choice of predicates.

A *corpus* $\mathcal{C}$ is a finite set of methods. We use the notation $\mathcal{C}/n$ to denote the set of methods in $\mathcal{C}$ that have name n, but where the semantics generally differs; and similarly $\mathcal{C}/s$ denotes the subcorpus of $\mathcal{C}$ where all methods have semantics s irrespective of their name. $\mathcal{C}/n$ is called a *nominal* corpus, $\mathcal{C}/s$ a *semantic* corpus.

Let x denote either a name component n or a semantics component s of some method. If $x_1, \ldots, x_k$ are all values occurring in $\mathcal{C}$ for a component, then we can view $\mathcal{C}$ as factored into disjoint subcorpora based on these values,

$$\mathcal{C} = \mathcal{C}/x_1 \cup \cdots \cup \mathcal{C}/x_k. \tag{10.1}$$

10.3.1.1 Corpus semantics and entropy

We repeat some information-theoretical concepts [5]. A *probability mass function* $p(x)$ is such that *a*) for all $i = 1, \ldots, k$ it holds that $0 \leq p(x_i) \leq 1$; and *b*) $\sum_{i=1}^{k} p(x_i) = 1$. Then $p(x_1), \ldots, p(x_k)$ is a *probability distribution*. From Equation (10.1) we observe

that the following defines a probability mass function:

$$p(\mathcal{C}/x) \stackrel{\text{def}}{=} \frac{|\mathcal{C}/x|}{|\mathcal{C}|}$$

We write p^N for the nominal probability mass function based on name factoring $\mathcal{C}/n$ and Equation (10.1), and p^S for the semantic version.

We define the semantics $[\![\mathcal{C}]\!]$ of a corpus $\mathcal{C}$ in terms of the distribution defined by p^S:

$$[\![\mathcal{C}]\!] \stackrel{\text{def}}{=} p(\mathcal{C}/s_1) \dots p^n(\mathcal{C}/s_k)$$

where we assume that $s_1, \dots, s_k$ are all possible semantic objects in $\mathcal{C}$ as in Equation (10.1). Of particular interest is the semantics of a nominal corpus; we therefore write $[\![n]\!]$ as a shorthand for $[\![\mathcal{C}/n]\!]$ when $\mathcal{C}$ is obvious from the context. This is what we intuitively refer to as "the meaning of n".

Using the probability mass function, we introduce a notion of entropy for corpora—similar to Shannon entropy [5].

$$H(\mathcal{C}) \stackrel{\text{def}}{=} -\sum_{x \in \chi} p(\mathcal{C}/x) \log_2 p(\mathcal{C}/x)$$

where we assume $0 \log_2 0 = 0$. We write $H^N(\mathcal{C})$ for the *nominal entropy* of $\mathcal{C}$, in which case χ denotes the set of all names in $\mathcal{C}$; and $H^S(\mathcal{C})$ for *semantic entropy* of $\mathcal{C}$, where χ denotes the set of all semantics. The entropy $H^S(\mathcal{C})$ is a measure of the semantic diversity of $\mathcal{C}$: High entropy means high diversity, low entropy means few different method implementations. Entropy $H^N(\mathcal{C})$ has the dual interpretation.

Entropy is particularly interesting on subcorpora of $\mathcal{C}$. The nominal entropy of a semantic subcorpus, $H^N(\mathcal{C}/s)$, measures the consistency in the naming methods with profile s in $\mathcal{C}$. The semantic entropy of a nominal subcorpus, $H^S(\mathcal{C}/n)$ measures the consistency in the implementation of name n. The nominal entropy of a nominal subcorpus is not interesting as it is always 0. The same holds for the dual concept. When there can be no confusion about $\mathcal{C}$, we speak of the nominal entropy of a profile s,

$$H^N(s) \stackrel{\text{def}}{=} H^N(\mathcal{C}/s)$$

and similarly for the dual concept $H^S(n)$.

Nominal entropy can be used to compare profiles. A profile with comparatively low nominal entropy indicates an implementation that tends to be consistently named. A profile with comparatively high nominal entropy indicates an ambiguous implementation. An obvious example of the latter is the empty method.

We can also compare the semantic entropy of names. A name with comparatively low semantic entropy implies that methods with that name tend to be implemented using a few, well-understood "cliches". A name with comparatively high semantic entropy implies that programmers cannot agree on what to call such method implementations (or that the semantics are particularly ill-suited at capturing the nature of the name).

We define aggregated entropy of corpus $\mathcal{C}$ as follows.

$$H_{agg}(\mathcal{C}) \stackrel{\text{def}}{=} \frac{\sum_{x \in \chi} |\mathcal{C}/x| H(\mathcal{C}/x)}{|\mathcal{C}|}$$

again leading to nominal and semantic notions of aggregated entropy, $H^N_{agg}(\mathcal{C})$ and $H^S_{agg}(\mathcal{C})$. These notions lets us quantify the overall entropy of subcorpora in $\mathcal{C}$, weighing the entropy of each subcorpus by its size.

10.3.1.2 Semantic cliches

When a method semantics is frequent in a corpus we call the semantics a semantic cliche, or simply a cliche, for that corpus. When a cliche has many different names we call it an unnameable cliche. Formally, a method semantics s is a *semantic cliche* for a corpus $\mathcal{C}$ if the prevalence of s in $\mathcal{C}$ is above some threshold value ϕ_{cl},

$$\frac{|\mathcal{C}/s|}{|\mathcal{C}|} > \phi_{cl}. \tag{10.2}$$

Furthermore, s is an *unnameable semantic cliche* if it satisfies the above, and in addition the nominal entropy of corpus $\mathcal{C}/s$ is above some threshold value H^N_{cl}, $H^N(\mathcal{C}/s) > H^N_{cl}$.

10.3.2 Semantic model

There are many ways of modelling the semantics of Java methods. For the purpose of comparing method names to implementations, we note one desirable property in particular. While the set of possible method implementations is practically unlimited, the set of different semantics in the model should both be finite and treat implementations that are essentially the same as having the same semantics. This is important, since each $\mathcal{C}/s$ should be large enough so that it is meaningful to speak of consistent or inconsistent naming of the methods in $\mathcal{C}/s$. This ensures that we can judge whether or not methods with semantics s are consistently named.

Some candidates for modelling method semantics are opcode sequences, abstract syntax trees and execution trace sets. However, we find these to be ill suited for our analysis: they do not provide a radical enough abstraction over the implementation. Therefore, we choose to model method semantics using an abstract language of bit vectors, as defined in Section 10.3.1.

Attributes. The abstract language relies on a set of predicates defined on Java bytecode. We refer to such predicates as *attributes* of the method implementation. Here we select and discuss the attributes we use, which yield a particular abstract language.

Individual attributes cannot distinguish perfectly between verbs. Rather, we expect to see trends when considering the probability that methods in each nominal corpus $\mathcal{C}/n$ satisfy each attribute. Furthermore, we note that 1) there might be names that are practically indistinguishable using bytecode predicates alone, and 2) some names are synonyms, and so *should* be indistinguishable.

Returns void	Returns field value
Returns boolean	Returns created object
Returns string	Runtime type check
No parameters	Creates custom objects[a]
Reads field	Contains loop
Writes field	Method call
Writes parameter value to field	Returns call result
Throws exceptions	Same verb call
Parameter value passed to method call on field value	

[a] A custom object is an instance of a type not in the `java.*` or `javax.*` namespaces.

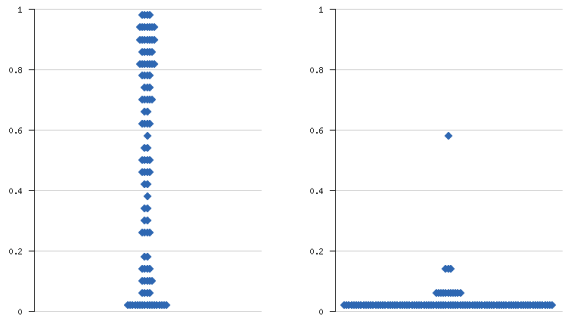
Table 10.1: Attributes.

Useful attributes. Intuitively, an attribute is *useful* if it helps distinguish between verbs. In Section 10.3.1, we noted that a verb might influence the probability that the predicate of an attribute is satisfied. Useful attributes have the property that this influence is significant. Attributes can be *broad* or *narrow* in scope. A broad attribute lets us identify larger groups of verbs that are aligned according to the attribute. A narrow attribute lets us identify smaller groups of verbs (sometimes consisting of a single verb). Both can be useful. The goal is to find a collection of attributes that together provides a good distinction between verbs.

Chosen attributes. We hand-craft a list of attributes for the abstract method semantics. An alternative would be to generate all possible simple predicates on bytecode instructions, and provide a selection mechanism to choose the "best" attributes according to some criterion. However, we find it useful to define predicates that involve a combination of bytecodes, for instance to describe control flow or subtleties in object creation. We deem it impractical to attempt a brute force search to find such combinations, and therefore resort to subjective judgement in defining attributes. To ensure a reasonable span of attributes, we pick attributes from the following categories: *method signature*, *object creation*, *data flow*, *control flow*, *exception handling* and *method calls*. The resulting attributes are listed in Table 10.1.

Probability distribution. The probability distribution for an attribute indicates if and how an attribute distinguishes between verbs. To illustrate, Figure 10.2 shows the probability distribution for two attributes: **Returns void** and **Writes parameter value to field**. Each dot represents the p^v for a given verb v, where v is a "common verb", as defined in Section 10.4.2. **Returns void** is a broad attribute, that distinguishes well between larger groups of verbs. However, there are also verbs that are ambiguous with respect to the attribute. By contrast, **Writes parameter value to field** is a narrow attribute. Most verbs have a very low probability for this attribute, but there is a single verb which stands out with a fairly high probability: this verb is `set`, which is rather unsurprising.

Critique. We have chosen a set of attributes for the semantic model based on our knowledge of commonly used method verbs in Java and how they are implemented.



(a) *Broad attribute:* **Returns void.**

(b) *Narrow attribute:* **Writes parameter value to field.**

Figure 10.2: Probability distribution for some attributes.

While all the attributes in the set are *useful* in the sense outlined above, we have no evidence that our set is "optimal" for the task at hand. There are two main problems with this.

First, we might have created an "unbalanced" set of attributes, meaning that we can have too many attributes capturing some kind of behaviour, such as object creation, and too few attributes capturing some other behaviour, such as exception handling. There might even be relevant behaviours that we have omitted altogether.

Second, we can construct many other attributes that could be used to distinguish between names; **Inverted method call**[4] and **Recursive call** are two candidates that we considered but rejected. The former is a narrow attribute that would help characterise `visit` methods, for instance. However, it turns out that `visit` is not ubiquitous enough to be included in our analysis (see Section 10.4.2); hence the attribute does not help in practice. The latter is simply too rarely satisfied to be very helpful.

The underlying problem is that there is no obvious metric by which to measure the quality of our attribute set. Arguably, the quality — or lack thereof — reveals itself in the results of our analysis.

10.3.3 Identifying synonyms

Intuitively, a verb n_1 is redundant if there exists another, more prevalent verb n_2 with *the same meaning*. It is somewhat fuzzy what "the same meaning" means. We define the meaning $[\![n]\!]$ of a verb n as the distribution of profiles in $\mathcal{C}/n$ (see Section 10.3.1). It is unlikely that the distributions for two verbs will be identical; however, some will be more similar than others. Hence we say that n_1 and n_2 have the same meaning if they are associated with sufficiently similar profile distributions.

We identify synonyms by investigating what happens when we merge the nominal corpora of two verbs. In other words, we attempt to eliminate one of the verbs, and

[4]By "inverted method call", we mean that the calling object is passed as a parameter to the method call.

investigate the effects on nominal and semantic entropy. If the effects are beneficial, we have identified a possible synonym.

The effects of synonym elimination. Elimination has two observable effects. First, there is a likely reduction in the aggregated nominal entropy H^N_{agg} of semantic corpora. The reason is that the nominal entropy of an individual semantic corpus is either *unaffected* by the elimination (if the eliminated verb is not used for any of the methods in the corpus), or it is *lowered*. Second, there is a likely increase in the aggregated semantic entropy H^S_{agg} of the nominal corpora — except for the unlikely event that the distribution of profiles is identical for the original corpora $\mathcal{C}/n_1$ and $\mathcal{C}/n_2$. How much H^S_{agg} increases depends on how semantically similar or different the eliminated verb is from the replacement verb. The increase in semantic entropy for the combined nominal corpus will be much lower for synonyms than for non-synonyms.

Optimisation strategy. When identifying synonyms, we must balance the positive effect on nominal entropy with the negative effect on semantic entropy. If we were to ignore the effect on semantic entropy, we would not be considering synonyms at all: simply to combine the two largest nominal corpora would yield the best effect. If we were to ignore the effect on nominal entropy, we would lose sight of the number of methods that are renamed. To combine a very large nominal corpus with a very small one would yield the best effect.

With this in mind, we devise a formula to guide us when identifying synonyms. A naive approach would be to demand that the positive effect on nominal entropy should simply be larger than the negative effect on semantic entropy. From practical experiments, we have found it necessary to emphasise semantic entropy over nominal entropy. That way, we avoid falsely identifying verbs with very large nominal corpora as synonyms. We therefore employ the following optimisation formula, which emphasises balance and avoids extremes, yet is particularly sensitive to increases in semantic entropy:

$$opt(\mathcal{C}) \stackrel{\text{def}}{=} \sqrt{4H^S_{agg}(\mathcal{C})^2 + H^N_{agg}(\mathcal{C})^2}$$

10.4 Software corpus

We have gathered a corpus of Java programs of all sizes, from a wide variety of domains. We assume that the corpus is large and varied enough for the code to be representative of Java programming in general. Table 10.2 lists the 100 Java applications, frameworks and libraries that constitute our corpus.

We filter the corpus in various ways to "purify" it:

- Omit compiler-generated methods (marked as **synthetic** in the bytecode).
- Omit methods that appear to have been code-generated.
- Omit methods without a common verb-name.
- Omit methods with unnameable semantics.

Desktop applications			
ArgoUML 0.24	Azureus 2.5.0	BlueJ 2.1.3	Eclipse 3.2.1
JEdit 4.3	LimeWire 4.12.11	NetBeans 5.5	Poseidon CE 5.0.1
Programmer tools			
Ant 1.7.0	Cactus 1.7.2	Checkstyle 4.3	Cobertura 1.8
CruiseControl 2.6	Emma 2.0.5312	FitNesse	JUnit 4.2
Javassist 3.4	Maven 2.0.4	Velocity 1.4	
Languages and language tools			
ANTLR 2.7.6	ASM 2.2.3	AspectJ 1.5.3	BSF 2.4.0
BeanShell 2.0b	Groovy 1.0	JRuby 0.9.2	JavaCC 4.0
Jython 2.2b1	Kawa 1.9.1	MJC 1.3.2	Polyglot 2.1.0
Rhino 1.6r5			
Middleware, frameworks and toolkits			
AXIS 1.4	Avalon 4.1.5	Google Web Toolkit 1.3.3	JXTA 2.4.1
JacORB 2.3.0	Java 5 EE SDK	Java 6 SDK	Jini 2.1
Mule 1.3.3	OpenJMS 0.7.7a	PicoContainer 1.3	Spring 2.0.2
Sun WTK 2.5	Struts 2.0.1	Tapestry 4.0.2	WSDL4J 1.6.2
Servers and databases			
DB Derby 10.2.2.0	Geronimo 1.1.1	HSQLDB	JBoss 4.0.5
JOnAS 4.8.4	James 2.3.0	Jetty 6.1.1	Tomcat 6.0.7b
XML tools			
Castor 1.1	Dom4J 1.6.1	JDOM 1.0	Piccolo 1.04
Saxon 8.8	XBean 2.0.0	XOM 1.1	XPP 1.1.3.4
XStream 1.2.1	Xalan-J 2.7.0	Xerces-J 2.9.0	
Utilities and libraries			
Batik 1.6	BluePrints UI 1.4	c3p0 0.9.1	CGLib 2.1.03
Ganymed ssh b209	Genericra	HOWL 1.0.2	Hibernate 3.2.1
JGroups 2.2.8	JarJar Links 0.7	Log4J 1.2.14	MOF
MX4J 3.0.2	OGNL 2.6.9	OpenSAML 1.0.1	Shale Remoting
TranQL 1.3	Trove	XML Security 1.3.0	
Jakarta commons utilities			
Codec 1.3	Collections 3.2	DBCP 1.2.1	Digester 1.8
Discovery 0.4	EL 1.0	FileUpload 1.2	HttpClient 3.0.1
IO 1.3.1	Lang 2.3	Modeler 2.0	Net 1.4.1
Pool 1.3	Validator 1.3.1		

Table 10.2: The corpus of Java applications and libraries.

Total methods	1.226.611
Non-synthetic	1.090.982
Hand-written	1.050.707
Common-verb name	818.503
Nameable semantics	778.715

Table 10.3: The effects of corpus filtering.

The purpose of the filtering is to reduce the amount of noise affecting our analysis. Table 10.3 presents some numbers indicating the size of the corpus and the impact of each filtering step.

10.4.1 Source code generation

Generation of source code represents a challenge for our analysis, since it can lead to a skewed impression of the semantics of a verb. In our context, the problem is this: a single application may contain a large number of near-identical methods, with identical verb and identical profile. The result is that the nominal corpus corresponding to the verb in question is "flooded" by methods with a specific profile, skewing the semantics of that corpus. Conversely, the semantic corpus corresponding to the profile in question is also "flooded" by methods with a specific verb, giving us a wrong impression of how methods with that profile are named.

To diminish the influence of code generation, we impose limits on the number of method instances contributed by a single application. By comparing the contribution from individual applications to that of all others, we can calculate an *expected* contribution for the application. We compare this with the *actual* contribution, and truncate the contribution if the ratio between the two numbers is unreasonable.

If the actual contribution is above some threshold T, then we truncate it to:

$$\max(T, \min(\frac{|\mathcal{C}_a/v|}{|\mathcal{C}/v|}, L\frac{|\mathcal{C}/v| - |\mathcal{C}_a/v|}{|\mathcal{C}| - |\mathcal{C}_a|}))$$

where L acts as a constraint on how much the contribution may exceed expectations.

Determining T and L is a subjective judgement, since we have no way of identifying false positives or false negatives among the method instances we eliminate. Our goal is to diminish the influence of code generation on our analysis rather than eliminate it. Therefore, we opt to be fairly lax, erring more on the side of false negatives than false positives. In our analysis, $T = 50$ and $L = 25$; that is, if some application contains more than 50 identical methods (n, s), we check that the number of identical methods does not exceed 25 times that of the average application. This nevertheless captures quite a few instances of evident code generation.

10.4.2 Common verbs

Some verbs are common, such as `get` and `set`, whereas others are esoteric, such as `unproxy` and `scavenge`. In this paper, we focus on the former and ignore the latter. There are several possible interpretations of *common*; two obvious candidates are *ubiquity* (percentage of applications) and *volume* (number of methods).

We choose ubiquity as our interpretation of *common*. Rudimentary grouping of verbs according to ubiquity is shown in Table 10.4. Since we are interested in the shared vocabulary of programmers, we restrict our analysis to the top three groups: *essential*, *core* and *extended*. The 102 verbs in these three groups cover nearly 77% of all methods (after filtering of generated code). Figure 10.3 shows a "word cloud" visualisation[5] of the common verbs.

[5]Generated by Wordle.net.

Vocabulary	*% Apps*	*Verbs*	*Methods*	*Example verbs*
essential	⟨ 90, 100]	7	50.73%	get, set, create
core	⟨ 75, 90]	21	13.26%	find, equals, parse
extended	⟨ 50, 75]	74	13.92%	handle, match, save
specific	⟨ 25, 50]	220	11.72%	sort, visit, refresh
narrow	⟨ 10, 25]	555	5.95%	render, shift, purge
marginal	⟨ 0, 10]	5722	4.43%	squeeze, unhook, animate

Table 10.4: Vocabularies.

Figure 10.3: The 102 most common verbs.

10.4.3 Unnameable cliches

Unnameable cliches, that is, method implementations that are common, yet inherently ambiguous, constitute noise for our analysis. We aim to reduce the impact of this noise by omitting methods whose implementations are unnameable cliches. The rationale is that the semantics of each verb will be more distinct without the noise, making it easier to compare and contrast the verbs.

In some cases, an implementation cliche may appear to be unnameable without being inherently ambiguous: rather, no generally accepted name for it has emerged. By applying canonicalisation through synonym elimination, the naming ambiguity can be reduced to normal levels. We must therefore distinguish between cliches that are *seemingly* and *genuinely* unnameable.

To identify implementation cliches, we use the concrete language $\mathcal{L}_{\text{Java}}$ (see Section 10.3.1). Table 10.5 shows unnameable cliches identified using Equation (10.2), with $\phi_{cl} = 500$ and $H^N_{cl} = 1.75$. We also include a reverse engineered example in a stylized Java source code-like syntax for each cliche.

To label the method implementations we apply f_{MD5}, which yields the MD5 digest of the opcodes for each implementation. Note that we include only the opcodes in the digest. We omit the operands to avoid distinguishing between implementations based on constants, text strings, the names of types and methods, and so forth. Hence f_{MD5} does abstract over the implementation somewhat. As a consequence, we cannot distinguish between, say, `this.`m`(`p`)` and p`.`m`(this)`: these are considered instances of the same cliche. Also, some cliches may yield the same example, since there are opcode sequences that cannot be distinguished when written as stylized source code.

Most of the cliches in Table 10.5 seem genuinely unnameable. Unsurprisingly, variations over delegation to other methods dominate. We cannot reasonably provide a name for such methods without considering the names of the methods being delegated to. There are also some examples of "unimplemented" methods; for instance { `throw new E();` } or the empty method { }. We believe that in many cases, the presence of these methods will be required by the compiler (for instance to satisfy some interface), but in practice, they will never be invoked.

Table 10.5 also contains three cliches that we deem only seemingly unnameable. This is based on a subjective judgement that they could be relatively consistently named, given a more well-defined vocabulary. These have been marked as being *retained*, meaning they are included in the analysis. The others are omitted.

10.5 Addressing synonyms

To address the problem of synonyms, we employ the formula *opt* from Section 10.3.3. We use *opt* to mechanically identify likely synonyms in the corpus described in Section 10.4, and then to attempt unsupervised elimination of synonyms.

10.5.1 Identifying synonyms

Compared to each of the common verbs in the corpus, the other verbs will range from synonyms or "semantic siblings" to the opposite or unrelated. To find the verbs that are semantically most similar to each verb, we calculate the value for *opt* when merging

Cliche	*# Methods*	H^N	*Retain*	*Top names*
{ super.m(); }	539	3.50		remove [10.6%], set [8.7%], insert [6.5%]
{ }	14566	3.40		set [18.1%], initialize [8.4%], end [7.8%]
{ this.m(); }	794	3.22		set [18.5%], close [7.2%], do [6.2%]
{ this.f.m(); }	2007	2.94		clear [20.7%], close [13.2%], run [11.7%]
{ return p; }	532	2.94		get [34.4%], convert [7.1%], create [4.7%]
{ super.m(p); }	742	2.69		set [32.2%], end [12.0%], add [9.4%]
{ throw new E(); }	3511	2.68		get [25.4%], remove [17.2%], set [14.8%]
{ this.f.m(); }	900	2.65		clear [28.2%], remove [16.1%], close [9.9%]
{ throw new E(s); }	5666	2.59		get [25.9%], set [22.3%], create [10.7%]
{ this.f.m(p); }	1062	2.48		set [39.2%], add [14.8%], remove [12.0%]
{ this.m(p); }	1476	2.45		set [24.4%], end [21.7%], add [14.2%]
{ return this.f.m(p); }	954	2.38		contains [25.9%], is [20.8%], equals [11.1%]
{ this.f.m(p_1, p_2); }	522	2.34		set [33.0%], add [17.2%], remove [13.0%]
{ return this.f.m(p); }	929	2.14		contains [28.3%], is [25.0%], get [11.1%]
{ return this.m(p); }	618	2.14		get [52.8%], post [8.4%], create [6.3%]
{ this.f = true; }	631	2.08	✓	set [48.5%], mark [12.8%], start [6.7%]
{ C.m(this.f); }	544	1.96		run [46.9%], handle [14.3%], insert [9.9%]
{ this.f.m(p); }	3906	1.92		set [36.8%], add [29.7%], remove [16.8%]
{ return new C(this); }	1540	1.87	✓	create [34.6%], get [25.7%], new [11.9%]
{ return this.m(); }	520	1.83		get [45.0%], is [20.0%], has [12.5%]
{ return false; }	6322	1.83	✓	is [52.8%], get [20.1%], has [7.3%]

Table 10.5: Semantic cliches with unstable naming.

the nominal corpus of each verb with the nominal corpus of each of the other verbs. The verbs that yield the lowest value for *opt* are considered synonym candidates.

It is more likely that two verbs are genuine synonyms if they reciprocally hold each other to be synonym candidates. When we identify such pairs of synonym candidates, we find that clusters emerge among the verbs, as shown in Figure 10.4.

Several of the clusters could be labelled, for instance as *questions*, *initialisers*, *factories*, *runners*, *checkers* and *terminators*. This suggests that these clusters have a "topic". It does not imply that all the verbs in each cluster could be replaced by a single verb, however. For instance, note that in the *factory* cluster, `create` and `make` are indicated as synonym candidates, as are `create` and `new`, but `new` and `make` are not. An explanation could be that `create` has a broader use than `new` and `make`.

We also see that there are two large clusters that appear to have more than one topic. We offer two possible explanations. First, polysemous verbs will tie together otherwise unrelated topics (see Section 10.2). In the largest cluster, for instance, we find a mix of verbs associated with I/O and verbs that handle collections. In this case, `append` is an example of a polysemous verb used in both contexts. Second, we may lack attributes to distinguish appropriately between the verbs.

10.5.2 Eliminating synonyms

To eliminate synonyms, we iterate over the collection of verbs. We greedily select the elimination that yields the best immediate benefit for *opt* in each iteration. We assume that beneficial eliminations will occur eventually, and that the order of eliminations is not important. We only label a synonym candidate as "genuine" if the value for *opt* decreases; the iteration stops when no more genuine candidates can be found. For

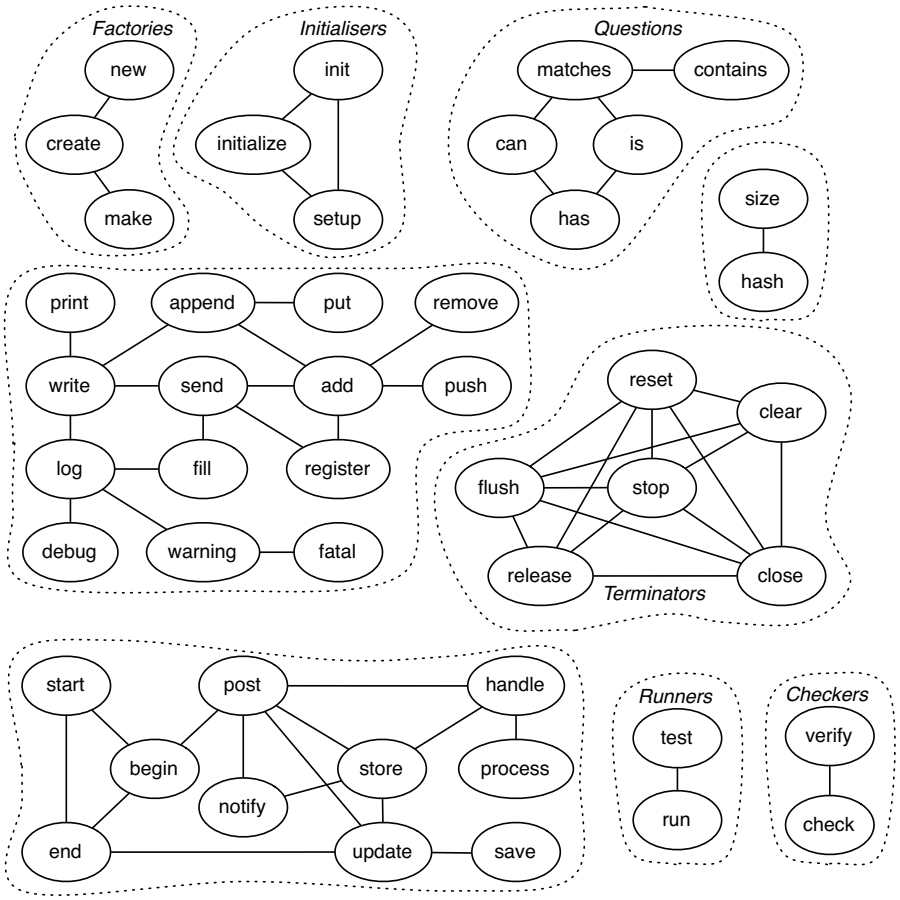


Figure 10.4: Clusters of synonym candidates. Clusters with a single topic are labelled.

comparison, we also perform manual elimination of synonyms, based on a hand-crafted list of synonym candidates.

The results of mechanical synonym elimination are shown in Table 10.6. Note that the input to the elimination algorithm is the output given by the preceding run of the algorithm. For the first run, the input is the original "purified" corpus described in Section 10.4, whereas for the second, the verb `has` has been eliminated, and the original nominal corpora for `has` and `is` have been merged.

The elimination of `has` is interesting: it is considered the most beneficial elimination by *opt*, yet as Java programmers, we would hesitate to eliminate it. The subtle differences in meaning between all "boolean queries" (`is`, `has`, `can`, `supports` and so forth) are hard to discern at the implementation level. Indeed, we would often accept method names with different verbs for the same implementation: `hasChildren` and `isParent` could be equally valid names. This kind of nominal flexibility is arguably beneficial for the readability of code.

It is easier to see that either `init` or `initialize` should be eliminated: there is no reason for the duplication. Eliminating `make` and using `create` as a canonical verb for factory methods also seems reasonable. Similarly, the suggestion to use `write` instead of `log` is understandable — however, one could argue that `log` is useful because it is more precise than the generic `write`.

There seems to be quite a few verbs for "termination code"; some of these verbs might be redundant. The unsupervised elimination process identifies `flush`, `stop` and `close` as candidates for synonym elimination. However, we find it unacceptable: certainly, `flush` and `close` cannot always be used interchangeably. In our coarse-grained semantic model, we lack the "semantic clues" to distinguish between these related, yet distinct verbs.

The suggestion to combine `add` and `remove` is also problematic, again showing that the approach has some limitations. Both `add` and `remove` typically involve collections of items, perhaps including iteration (which is captured by the **Contains loop** attribute). The crucial distinction between the two operations will often be hidden inside a call to a method in the Java API. Even if we were to observe the actual adding or removing of an item, this might involve incrementing or decrementing a counter, which is not captured by our model.

Table 10.7 shows the result of the manual elimination of synonyms. We note that only the elimination of `initialize` yields a decreased value for *opt* — apparently, we are not very good at manual synonym identification! However, it may be that the requirement that *opt* should decrease is too strict. Indeed, we find that many of our candidates are present in the clusters shown in Figure 10.4. This indicates that there is no deep conflict between our suggestions and the underlying data.

10.5.3 Canonicalisation

Overall, we note that our approach succeeds in finding relevant candidates for synonym elimination. However, it is also clear that the elimination must be supervised by a programmer. We therefore suggest using Figure 10.4 as a starting point for manual canonicalisation of verbs in method names. Canonicalisation should entail both eliminating synonyms and providing a precise definition, rationale and use cases for each verb.

Run	*Canonical (cv)*	*Old verbs*	$\|\mathcal{C}/cv\|$	*Sum*	ΔH^S_{agg}	ΔH^N_{agg}	Δopt
1	is	has+is	49041	6820+42221	0.00269	-0.02270	-0.01152
2	is	can+is	51649	2608+49041	0.00178	-0.01148	-0.00409
3	add	remove+add	43241	16172+27069	0.00667	-0.03004	-0.00237
4	init	initialize+init	11026	3568+7458	0.00149	-0.00743	-0.00126
5	close	stop+close	5025	1810+3215	0.00074	-0.00348	-0.00040
6	create	make+create	38140	4940+33200	0.00363	-0.01525	-0.00021
7	close	flush+close	5936	911+5025	0.00061	-0.00266	-0.00014
8	reset	clear+reset	5849	2901+2948	0.00100	-0.00421	-0.00007
9	write	log+write	13659	1775+11884	0.00131	-0.00547	-0.00004

Table 10.6: Mechanical elimination of synonyms.

Canonical (cv)	*Old verbs*	$\|\mathcal{C}/cv\|$	*Sum*	ΔH^S_{agg}	ΔH^N_{agg}	Δopt
clone	clone+copy	4732	2595+2137	0.00271	-0.00147	0.00979
execute	execute+invoke	4947	2997+1950	0.00197	-0.00229	0.00589
verify	check+verify	8550	7440+1110	0.00126	-0.00298	0.00223
stop	stop+end	4814	1810+3004	0.00126	-0.00283	0.00242
write	write+log+dump	15987	11884+1775+2328	0.00420	-0.01109	0.00635
start	start+begin	5485	4735+750	0.00081	-0.00200	0.00135
init	init+initialize	11026	7458+3568	0.00149	-0.00743	-0.00126
error	error+fatal	1531	1116+415	0.00027	-0.00088	0.00023
create	create+new+make	45565	33200+7425+4940	0.00901	-0.03588	0.00152

Table 10.7: Manual elimination of synonyms.

10.6 Related Work

Gil and Maman [9] introduce the notion of machine-traceable patterns, in order to identify so-called micro patterns; machine-traceable implementation patterns at the class level. When we model the semantics of method implementations using hand-crafted bytecode predicates, we could in principle discern "nano patterns" at the method implementation level. According to Gamma et al. [8], however, a pattern has four essential elements: name, problem, solution and consequences. Though we do identify some very commonly used implementation cliches, we do not attempt to interpret and structure these cliches. Still, Singer et al. [20] present their own expanded set of bytecode predicates under the label "fundamental nano patterns", where the term "pattern" must be understood in a broader, more colloquial sense.

Collberg et al. [4] present a large set of low-level statistics from a corpus of Java applications, similar in size to ours. Most interesting to us are the statistics showing k-grams of opcodes, highlighting the most commonly found opcode sequences. This is similar to the implementation cliches we find in our work. Unfortunately, the k-grams are not considered as logical entities, so a common 2-gram will often appear as part of a common 3-gram as well.

Similar in spirit to our work, Singer and Kirkham [21] find a correlation between certain commonly used type name suffixes and some of Gil and Maman's micro patterns. Pollock et al. [17] propose using "natural language program analysis", where natural language clues found in comments and identifiers are used to augment and guide program analyses. Tools for program navigation and aspect mining have been developed [19, 18] based on this idea. Ma et al. [15] exploit the fact that programmers

usually choose appropriate names in their code to guide searches for software artefacts.

The quality of identifiers is widely recognised as important. Deißenböck and Pizka [6] seek to formalise two quality metrics, *conciseness* and *consistency*, based on a bijective mapping between identifers and concepts. Unfortunately, the mapping must be constructed by a human expert. Lawrie et al. [13] seek to overcome this problem by deriving syntactic rules for conciseness and consistency from the identifiers themselves. This makes the approach much more applicable, but introduces the potential for false positives and negatives.

10.7 Conclusion and further work

The ambiguous vocabulary of verbs used in method names makes Java programs less readable than they could be. We have identified *redundancy*, *coarseness* and *vagueness* as the problems to address. In this paper, we focussed on *redundancy*, where more than one verb is used in the same meaning. We looked at the identification and elimination of synonymous verbs as a means towards this goal.

We found that we were indeed able to identify reasonable synonym candidates for many verbs. To select the genuine synonyms among the candidates without human supervision is more problematic. The abstract semantics we use for method implementations is sometimes insufficient to capture important nuances between verbs. A more sophisticated model that takes into account invoked methods, either semantically (by interprocedural analysis of bytecode) or nominally (by noting the names of the invoked methods) might overcome some of these problems. Realistically, however, the perspective of a programmer will probably still be needed. A more fruitful way forward may be to use the identified synonym candidates as a starting point for a manual process where a canonical set of verbs is given precise definitions, and the rest are discouraged from use.

Addressing the problem of *coarseness* is a natural counterpart to the topic of this paper. Coarseness manifests itself in polysemous verbs, that is, verbs that have more than a single meaning. Polysemous verbs could be addressed by investigating the semantics of the methods that constitute a nominal corpus $\mathcal{C}/n$. The intuition is that polysemous uses of n will reveal itself as clusters of semantically similar methods. Standard data clustering techniques could be applied to identify such polysemous clusters. If a nominal corpus were found to contain polysemous clusters, we could investigate the effect of renaming the methods in one of the clusters. This would entail splitting the original nominal corpus $\mathcal{C}/n$ in two, $\mathcal{C}/n$ and $\mathcal{C}/n'$. The effect of splitting the corpus could be measured, for instance by applying the formula *opt* from Section 10.3.3.

Bibliography

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs - 2nd Edition (MIT Electrical Engineering and Computer Science)*. The MIT Press, 2 edition, July 1996.

[2] K. Beck. *Implementation Patterns*. Addison-Wesley Professional, 2007.

[3] E. Collar and R. Valerdi. Role of software readability on software development cost. In *Proceedings of the 21st Forum on COCOMO and Software Cost Modeling, October 2006, Herndon, VA.*, 2006.

[4] C. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. *Software Practice and Experience*, 37(6):581–641, 2007.

[5] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. Wiley, 2nd edition, 2006.

[6] F. Deißenböck and M. Pizka. Concise and consistent naming. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005)*, pages 97–106. IEEE Computer Society, 2005.

[7] M. A. Eierman and M. T. Dishaw. The process of software maintenance: A comparison of object-oriented and third-generation development languages. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(1):33–47, 2007.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[9] J. Gil and I. Maman. Micro patterns in Java code. In R. E. Johnson and R. P. Gabriel, editors, *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), October 16-20, 2005, San Diego, CA, USA*, pages 97–116. ACM, 2005.

[10] E. W. Høst and B. M. Østvold. The programmer's lexicon, volume I: The verbs. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 193–202, Paris, France, 2007. IEEE Computer Society.

[11] E. W. Høst and B. M. Østvold. The Java programmer's phrase book. In *Proceedings of the 1st International Conference on Software Language Engineering (SLE 2008)*. Springer, 2008.

[12] E. W. Høst and B. M. Østvold. Debugging method names. In S. Drossopoulou, editor, *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, volume 5653 of *Lecture Notes in Computer Science*, pages 219–317. Springer, 2009.

[13] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), 27-29 September 2006, Philadelphia, Pennsylvania, USA*, pages 139–148. IEEE Computer Society, 2006.

[14] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.

[15] H. Ma, R. Amor, and E. D. Tempero. Indexing the Java API using source code. In *Proceedings of the 19th Australian Software Engineering Conference (ASWEC 2008), March 25-28, 2008, Perth, Australia*, pages 451–460. IEEE Computer Society, 2008.

[16] R. C. Martin. *Clean Code*. Prentice Hall, 2008.

[17] L. L. Pollock, K. Vijay-Shanker, D. Shepherd, E. Hill, Z. P. Fry, and K. Maloor. Introducing natural language program analysis. In M. Das and D. Grossman, editors, *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007), San Diego, California, USA, June 13-14, 2007*, pages 15–16. ACM, 2007.

[18] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development (AOSD 2007)*, pages 212–224, New York, NY, USA, 2007. ACM.

[19] D. Shepherd, L. L. Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In R. E. Filman, editor, *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006), Bonn, Germany, March 20-24, 2006*, pages 3–14. ACM, 2006.

[20] J. Singer, G. Brown, M. Lujan, A. Pocock, and P. Yiapanis. Fundamental nanopatterns to characterize and classify Java methods. In *Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications (LDTA 2009)*, pages 204–218, 2009.

[21] J. Singer and C. Kirkham. Exploiting the correspondence between micro patterns and class names. In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008)*, pages 67–76, Beijing, China, 2008. IEEE Computer Society.

[22] L. Steels. The recruitment theory of language origins. In C. Lyon, C. L. Nehaniv, and A. Cangelosi, editors, *Emergence of Language and Communication*, pages 129–151. Springer, 2007.

[23] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.